The Effect of Deceptive Idleness on Disk Schedulers

by

Sitaram Iyer

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree Master of Science

Approved, Thesis Committee:

Dr. Peter Druschel (Chair),
Associate Professor,
Computer Science

Dr. Willy Zwaenepoel,
Professor,
Computer Science

Dr. Edward W. Knightly,
Assistant Professor,
Electrical and Computer Engineering

Houston, Texas
April, 2001
The Effect of Deceptive Idleness on Disk Schedulers

Sitaram Iyer

Abstract

Disk schedulers in operating systems are generally work-conserving; they schedule a request immediately after the previous request has finished. Such schedulers need multiple outstanding requests to make good decisions. Unfortunately, many applications issue synchronous, almost-continuous streams of read requests. This forces the scheduler into making decisions too early, falsely assuming that the process has become momentarily idle. This phenomenon of deceptive idleness causes significant degradation in performance and quality of service objectives on current systems. We solve deceptive idleness by designing and implementing a transparent, non-work-conserving scheduling framework for various scheduling policies. We evaluate this solution on microbenchmarks and real workloads, and observe large benefits. The Apache webserver delivers 56% and 16% more throughput for two configurations. The Andrew Benchmark runs faster by 8% (54% for the read-intensive phase). Variants of the TPC-B database benchmark exhibit improvements between 4% and 60%. Proportional-share schedulers become empowered to efficiently deliver application-desired proportions.
Acknowledgments

I wish to thank a number of people who have contributed to the making of this thesis. First and foremost, I thank my advisor Dr. Peter Druschel for his visionary supervision throughout the project, and the freedom and flexibility that he allowed me at every stage. The driving force can be rightly attributed to his adamant insistence, especially during the formative days, that this project can actually be pulled off. I cannot sufficiently thank Juan Navarro and Karthick Rajamani for the marvellous feedback-based sounding boards that they have both been. Mohit Aron gave me useful advice regarding the choice and usage of timing mechanisms. I had countless lunchtime conversations with Sanjeeb Dash about the material in this thesis. Martin DeMello coined the phrase “deceptive idleness” after hearing a boring description of the same; it sounds so much cuter than the originally conceived “apparent idleness”. Sameer Siruguri helped print and submit the thesis when I wasn’t physically around. Both Rice University and MIT have provided computational resources used for this project; I thank them for that. Finally, I express sincere thanks to my family for their continued support over the years.
Contents

Abstract ii
Acknowledgments iii
List of Illustrations vii

1 Introduction 1
1.1 Thesis contributions .............................................. 4
1.2 Structure of this thesis .............................................. 4

2 Background 5
2.1 Basics of disk scheduling ........................................ 5
2.1.1 The scheduling framework .................................... 7
2.1.2 Seek-reducing scheduling policies .......................... 8
2.1.3 Proportional-share scheduling policies ...................... 9
2.2 Terminology ........................................................ 10
2.3 The underlying problem: deceptive idleness ................. 12

3 Sources of deceptive idleness 13
3.1 Request dependencies ............................................. 13
3.2 Resource contention .............................................. 15
3.3 Experimental characterization of deceptive idleness .......... 16
3.3.1 No deceptive idleness ........................................ 16
3.3.2 Explicit request dependencies ............................... 17
3.3.3 Consequences of metadata sharing ........................ 18
3.3.4 Resource contention for memory ........................... 18
4 Effects of deceptive idleness

4.1 Seek-reducing disk schedulers ........................................ 20
4.2 Proportional-share disk schedulers .................................... 22
  4.2.1 Experimental characterization on Stride ......................... 23
  4.2.2 More than two resource containers ............................... 26
  4.2.3 Simultaneous seek reduction ..................................... 29

5 Workarounds for deceptive idleness .................................. 30

  5.1 Application-driven prefetch ......................................... 30
  5.1.1 Multi-process architecture ....................................... 31
  5.1.2 Asynchronous I/O .................................................. 31
  5.1.3 An mmap/mincore/read interface .................................. 33
  5.1.4 Metadata caching .................................................. 33

  5.2 Kernel-driven prefetch ............................................... 34
  5.2.1 Async prefetch may be completely impossible .................. 35
  5.2.2 Async prefetch may be impossible from the OS ................ 35
  5.2.3 Async prefetch may need large files to detect ............... 36
  5.2.4 Async prefetch may not be implemented: practical difficulties 36
  5.2.5 Summary of this section ......................................... 39

6 Non-work-conserving scheduler solution .......................... 40

  6.1 The NWCS waiting mechanism ....................................... 41
  6.2 Process-granularity assumptions .................................... 44
    6.2.1 Synchronous requests issued by processes .................... 44
    6.2.2 Request homogeneity within a process ....................... 45
  6.3 Seek reducing schedulers ......................................... 45
    6.3.1 Variant: ASPTF: Aged SPTF .................................. 47
    6.3.2 Variant: C-LOOK: Cyclic look ............................... 48
  6.4 Proportional-share schedulers .................................... 48
6.4.1 Variant: YFQ: Yet-another Fair Queueing .................. 49
6.5 Heuristic combination ............................................. 49
6.6 Real-time schedulers .............................................. 50
6.7 Implementation notes ............................................. 51
   6.7.1 Tagged queueing in SCSI controllers ..................... 52
6.8 Potential improvements ........................................... 54
   6.8.1 Accumulate more statistics .................................. 54
   6.8.2 Fix the process-granularity assumptions ................. 54

7 Experimental evaluation ................................. 56
    7.1 Microbenchmarks ............................................. 56
    7.1.1 Different access patterns ................................. 57
    7.1.2 Varying thinktimes ....................................... 58
    7.1.3 Proportional-share scheduling ............................ 61
    7.2 Real workloads .............................................. 62
    7.2.1 The Andrew Benchmark .................................... 63
    7.2.2 The Apache webserver .................................... 64
    7.2.3 The GnuLD linker ......................................... 66
    7.2.4 The TPC-B database benchmark ............................ 67

8 Related work ............................................... 69
    8.1 Scheduling algorithms ....................................... 69
    8.2 Improved I/O processing outside the scheduling policy .... 72
    8.3 Other scheduling domains .................................... 73

9 Conclusion ................................................. 75

Appendix: Detailed NWCS pseudocode ........................ 75

Bibliography .................................................. 80
Illustrations

2.1 State transitions for a conventional scheduler ........................................ 7
2.2 A traditional work-conserving scheduling framework ............................. 8

3.1 No deceptive idleness ............................................................................... 16
3.2 Deceptive idleness due to explicit request dependencies ......................... 17
3.3 Deceptive idleness due to shared metadata ............................................ 18
3.4 Deceptive idleness due to memory contention ......................................... 19

4.1 Deceptive Idleness on a seek reducing scheduler: (a) effective seek
reduction, and (b) thrashing due to deceptive idleness ................................. 21
4.2 Deceptive Idleness on a proportional-share scheduler: (a) effective 1:2
allocation, and (b) skewed 1:1 allocation due to deceptive idleness .......... 22
4.3 Proportional-share scheduler delivering desired proportions .................... 23
4.4 Deceptive idleness due to explicit request dependencies ........................ 24
4.5 Deceptive idleness due to shared metadata ............................................ 25
4.6 Deceptive idleness due to memory contention ....................................... 25
4.7 Lottery scheduler behaving as desired .................................................... 27
4.8 Lottery scheduler achieving skewed proportions due to deceptive
idleness ........................................................................................................ 28

5.1 Impact of asynchronous prefetch on seek-reducing scheduler
performance ................................................................................................... 38
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1 NWCS system architecture</td>
<td>42</td>
</tr>
<tr>
<td>6.2 State transition diagram for the NWCS waiting mechanism</td>
<td>43</td>
</tr>
<tr>
<td>7.1 Different access patterns</td>
<td>57</td>
</tr>
<tr>
<td>7.2 Increasing thinktimes for both processes</td>
<td>59</td>
</tr>
<tr>
<td>7.3 Increasing thinktimes for one process</td>
<td>60</td>
</tr>
<tr>
<td>7.4 Adversary application</td>
<td>61</td>
</tr>
<tr>
<td>7.5 Seek-reducing proportional-share scheduler</td>
<td>62</td>
</tr>
<tr>
<td>7.6 Andrew Benchmark</td>
<td>63</td>
</tr>
<tr>
<td>7.7 The Apache webserver</td>
<td>65</td>
</tr>
<tr>
<td>7.8 The GNU Linker</td>
<td>66</td>
</tr>
<tr>
<td>7.9 The TPC-B database benchmark</td>
<td>67</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Disk scheduling has been an integral part of operating system functionality since the early days. Many scheduling algorithms [SC090, WGP94, BBG+99, HC00, JW91] have been proposed for different purposes. The earlier ones targetted seek reduction to achieve high throughput and low response time. More recently, differentiated quality of service has become an important goal, and scheduling policies have been proposed to provide proportional-share and real-time disk service to applications.

These policies essentially define the manner in which available disk requests are to be reordered to meet the desired objective. Most operating systems enclose such policies in a work-conserving scheduling framework, which selects and schedules requests as soon as (or before) the previous request has finished. At the moment of request selection (i.e. at a decision point), a scheduler requires multiple outstanding requests from active applications to make an effective decision. This means that a work-conserving scheduler becomes capable of providing the desired type of service only to applications that actively and continuously generate disk requests.

Unfortunately, a common trend in application behaviour is to issue synchronous disk requests. Intrinsic dependencies between successive requests may force a process to issue a request only a few hundred microseconds after it has seen the outcome of the previous one. Often, many such processes on the system concurrently issue streams of synchronous disk requests. In such situations, a typical work-conserving scheduler becomes inherently incapable of consecutively servicing multiple requests from any process. Instead, at each decision point, it is forced to multiplex between requests issued by different processes.
Viewing this from a system-wide perspective, schedulers are expected to have an accurate understanding of whether a request-issuing entity, such as a process, is busy or idle. Concurrent streams of synchronous requests induce a condition wherein the scheduler assumes the last request issuing process to have become momentarily idle at decision points. In reality, the process has just not had the chance to issue the subsequent request before the scheduler makes its decision. The scheduler thus suffers from the phenomenon of *deceptive idleness*.

A seek reducing scheduler may try to exploit spatial locality between successive requests issued by each process. However, deceptive idleness can cause an overall degradation of performance due to inevitable head movement between requests issued by different processes. Seek-reducing schedulers may thus yield low performance under the influence of deceptive idleness. A proportional-share scheduler may be required to deliver proportions such as 2:1 between processes. To achieve this, it becomes inherently required to service two or more requests successively from the first process. Deceptive idleness may prevent this, and cause a significant deviation of achieved proportions from the assigned ones. In general, a work-conserving implementation of a scheduling policy reorders the available requests correctly, but can fail to meet overall system-wide objectives.

Some of these effects feature in commonly used systems and are somewhat well-known, while some others have not been examined previously. This thesis takes a more holistic approach towards the analysis of this problem. We demonstrate how deceptive idleness often happens in general-purpose operating systems for a variety of reasons, ranging from naively chosen programming paradigms to inevitable metadata dependencies. We then characterize its impact on two somewhat representative classes of disk schedulers: seek-reducing and proportional-share schedulers. These effects turn out to be fundamentally different in some ways, and well-known workarounds for some schedulers are ineffective for others.

We discuss workarounds based on prefetching, both from the application and from
the kernel. Upon detailed examination, both these prove to be insufficient in many cases. Kernel-level prefetching requires precise prediction of subsequent requests, and incurs heavy costs on error. The more powerful application-based workarounds are not always possible either, and are obviously undesirable in situations where applications are expected to run unmodified.

We therefore propose a simple, transparent solution to deceptive idleness. This is based on a non-work-conserving scheduling (NWCS) framework, i.e. one that selectively lets the disk remain unutilized for short, controlled periods of time at decision points. Additional requests from the last request issuing process may arrive in this period, giving the scheduler the opportunity to make better decisions and transparently eliminate deceptive idleness. NWCS can be implemented in a general-purpose operating system without altering the actual scheduling policy. It automatically supplements any prefetching mechanisms already in place, along with any application-level measures that prevent deceptive idleness. It thus incurs minimal performance overhead (less than 1%) under normal conditions, and substantially improves throughput and adherence to QoS objectives whenever deceptive idleness sets in.

We evaluate this solution on a FreeBSD-4.0 system, using a variety of microbenchmarks and real workloads. Microbenchmarks indicate throughput improvements of up to 4 times, under an extensive range of workload characteristics. Real workloads often embody more random access, and thus yield smaller, though significant benefits. Loss in utilization due to the non-work-conserving scheduler is nearly always exceeded by throughput improvement due to seek optimization. By capitalizing on seek reduction within files, a disk-intensive Apache webserver is found to deliver 56% and 16% more throughput for two configurations (mmap and read). The Andrew Benchmark runs faster by 8%, (with its synchronous read intensive phase by 54%), by reducing seeks both within and between files. Variants of the TPC-B database benchmark exhibit improvements between 4% and 60%. Proportional-share schedulers like Stride become empowered to accurately distribute disk resources according
to the assigned proportions, even to processes that synchronously issue disk requests.

1.1 Thesis contributions

The contributions of this thesis are twofold:

- Identification and thorough analysis of a fundamental problem that affects the behaviour of almost all disk schedulers. Demonstration of how existing mechanisms are not sufficient to solve it in all situations.

- Proposal and experimental evaluation of a new non-work-conserving scheduling framework to address this problem. This framework is shown to be qualified for implementation in general-purpose operating systems, where it supplements the workarounds already present.

1.2 Structure of this thesis

We begin with a discussion of some background material in chapter 2 that would help in understanding the rest of this thesis. The chapters following that essentially constitute different cross-sections into the same basic problem. Chapter 3 talks about various reasons and scenarios that cause deceptive idleness. After that, chapter 4 analyzes its impact on a variety of disk scheduler types, at two levels of detail. Having sufficiently explored the problem, we discuss a variety of workarounds and solutions in chapter 5. We elaborate upon and evaluate our proposed non-work-conserving scheduling framework in chapter 6. Finally, we present some related work in this area and conclude.

This thesis differs from the associated paper [ID01b], primarily in its systematic approach to development of the problem. It describes all possible sources of deceptive idleness, and separately analyzes their effects on a variety of disk schedulers. Reading the paper first is recommended for quick understanding. The project website is hosted at [ID01a].
Chapter 2

Background

The performance gap between disks and other components of a computer system is large, and has been steadily widening over the years. Disk speeds are growing by only about 7\% a year, and are not keeping pace with the 55\% annual increase in CPU speeds. A substantial fraction of modern day systems are disk intensive, such as database systems, file servers, large working-set web and ftp servers, and various types of scientific computation. New application types like multimedia clients and servers are driving the need for differentiated quality of service in the disk subsystem. It is thus becoming increasingly important for modern operating systems to provide high performance and accurate quality of service to applications.

The disk subsystem in an operating system can be viewed as a channel for disk requests from applications to the disk controller. A disk scheduler is a vital component of the disk subsystem, whose goal is to reorder these requests. This chapter presents an overview of disk schedulers as implemented in most operating systems.

2.1 Basics of disk scheduling

Consider a single physical disk attached to a computer system. Applications invoke system calls like \texttt{read} and \texttt{write} to gain access to disk blocks. These are sometimes serviced from the filesystem cache that the kernel maintains; misses in this cache become disk requests. Applications may similarly map files into memory, and somehow access this memory region. This causes page faults that may sometimes also result in disk I/O. Most operating systems perform background maintenance tasks such as swap-space management and disk defragmentation. These could lead to disk I/O that
is not directly initiated by applications.

These requests are serviced by the disk one after another, in some sequence. This sequence is determined by the disk scheduling policy, and thus, by the disk scheduler implementation. The disk scheduler is an artifact of the operating system that receives multiple requests, queues them internally in some fashion, and dispatches them one by one to the disk driver for (usually) immediate service.

Disk scheduling is fundamentally different in many ways from the related problem of CPU scheduling. Disk scheduling is explicitly request-driven. These requests are associated with two metrics: a request size, which is known in advance, and a request service time calculated in finish. These respectively result in performance measures like throughput in MB/s and disk utilization in fraction of disk time spent servicing requests. In contrast, CPU quanta are only associated with the time spent running on the processor. Disk scheduling is a non-preemptive discipline, i.e., a request, once dispatched to the disk for service, cannot be withdrawn. In CPU scheduling, the processor can be relinquished early or the quantum forcibly preempted. Context switching overheads for disk take the form of seek and rotational latencies, and may range from very small to very large, depending on request placement. Context switching in CPU is moderately expensive, and does not possess as much variation. Such differences lead to problems unique to disk scheduling, such as deceptive idleness.

A disk scheduler is structurally composed of two parts: the scheduling framework, and an implementation of the scheduling policy. The framework provides the required interface between the scheduling policy and the rest of the kernel. The policy forms the actual implementation of some kind of request queueing, and thus provides a set of scheduler-specific methods invoked by the framework. The idea is to keep the framework constant between individual schedulers, while changing just these methods.

A scheduler is said to be work-conserving if it never lets the disk idle whenever there are requests waiting to receive service. A non-work-conserving scheduler may

---

¹not the fraction of time spent doing useful work.
allow the disk to become idle, despite the presence of pending requests. The next section examines the structure and functioning of a traditional work-conserving disk scheduling framework.

2.1.1 The scheduling framework

The scheduling framework provides the infrastructure that receives requests from upper layers of the kernel, dispatches requests to the disk driver, handles completion of request service, etc. We slightly overstate its importance, since this is the part of the scheduler that we shall later be most concerned about.

![State transitions for a conventional scheduler](image)

Figure 2.1: State transitions for a conventional scheduler

The scheduler operates between two states: IDLE and BUSY, thus reflecting the running state of the disk (figure 2.1). The disk is initially idle. Requests arrive at the scheduler at any time; these are enqueued into the request pool using the scheduler-specific `sched_enqueue(req)` method. If the disk was idle at this moment, the work-conserving scheduling framework is compelled to issue a request to the disk immediately, thus calling `schedule`. This moment in time is a decision point. The scheduler chooses a request using the `sched_choose()` method, removes the request from the pool using `sched_dequeue(req)`, switches state to BUSY, and dispatches the request to the disk driver for immediate service\(^2\). When a request finishes receiving service, the scheduler state is switched back to IDLE and the `sched_finish(req)`

\(^2\)SCSI disks implement tagged queueing, wherein multiple requests are scheduled to the controller, internally seek optimized and then serviced. We discuss this later.
method is invoked if the scheduling policy wishes to be notified of this event. Then
\texttt{schedule} is called again if any more requests are found pending (figure 2.2).

```c
enum STATE { IDLE, BUSY } state = IDLE;
integer pending = 0;

issue(new): // invoked by upper layers of the kernel
    sched_enqueue(new);
    pending++;
    if (state != BUSY) schedule();

schedule(): // local function
    assert(pending > 0);
    assert(state == IDLE);
    next = sched_choose();
    sched_dequeue(next);
    pending--;
    state = BUSY;
    // send to disk driver for immediate service
    PERFORM_DISK_IO(next);

finish(req): // called from the disk interrupt handler
    assert(state == BUSY);
    state = IDLE;
    sched_finish(req); // if the scheduler wants to know
    if (pending) schedule();
```

Figure 2.2: A traditional work-conserving scheduling framework

2.1.2 Seek-reducing scheduling policies

The simplest disk scheduling policy is First Come First Served (FCFS), wherein
\texttt{sched_enqueue} and \texttt{sched_dequeue} just maintain a FIFO queue. However, this policy
is efficient only for very light workloads, with little concurrency between requests.
Magnetic disks with movable heads traditionally incur large overheads for positioning
the disk head over the desired sector. In order to amortize this cost over larger data
transfers, disk schedulers implement different kinds of seek reduction policies.

To give an example, a 7200 rpm IBM Deskstar 34GXP disk sustains a continuous
21 MB/s of sequential data transfer, but only about 5 MB/s of random-access 64k
block transfer. This is because a 64k chunk takes 3ms to read from this disk, as compared to an average seek time of about 9ms.

The underlying idea in all seek-reducing schedulers is thus to capitalize on spatial locality of disk access. If a request issuer generates requests that are targetted at nearby disk blocks, the scheduler should be able to service multiple requests with very little head seek and platter rotation. Schedulers take a variety of approaches to implement this, often without starving any requests.

The Shortest Positioning-Time First (SPTF) policy greedily schedules the available request with the smallest head repositioning time. To eliminate the possibility of starving distant requests, Aged-SPTF (ASPTF) gives increasing priority to requests that have been kept pending for long. A scheduling policy commonly implemented in UNIX-based systems is the Elevator algorithm (C-LOOK), wherein the disk head traverses in only one direction, servicing all available requests on its way. On reaching the last request, it quickly moves to the first request and starts over [WGP94, JW91, SCO90]. Chapter 8 discusses these in greater detail.

2.1.3 Proportional-share scheduling policies

With the evolution of the Internet, there grew an increasing demand for differentiated quality of service support in operating systems. This broadly translates to suitable management of all system resources, including disk service. Disk schedulers are thus becoming required to provide more functionality than just raw performance.

The central theme in differentiated QoS is to provide support for multiple service classes, and allow applications associated with each service class to be treated according to some service agreement. Traditional operating systems multiplex system resources among resource principals such as processes and threads. Banga et al. [BDM99] proposed the resource container abstraction that separates the notion of a resource principal from a process or a thread, thus providing the potential for fine-grained resource management. We use resource containers as our service class
abstraction.

Among various types of differentiated service, this thesis mainly examines the impact of deceptive idleness on proportional-share scheduling. The service agreement in proportional-share disk schedulers assumes the form of a pre-assigned set of proportions; these are expected to be matched by the amount of disk service delivered to active classes. Several proportional-share scheduling algorithms have been proposed; some of these are described in chapter 8.

This thesis uses the Stride scheduler [Wal95, WW95], trivially adapted for disk scheduling. This algorithm performs deterministic proportional-time scheduling, i.e. it proportionally distributes disk service time (and thus disk utilization) among resource containers on a fine granularity. It maintains a separate virtual clock per resource container, and increments it on completion of a request attached to that container. The increment is proportional to the request service time, and is inversely related to the share assigned to the container. At decision points, a request is selected from the container with the smallest virtual clock. A container remaining idle for an extended period of time explicitly leaves; upon returning, it is assigned a virtual clock equal to a global clock that corresponds to the smallest virtual clock of all active containers.

A direct adaptation of Stride to disk scheduling would generally incur significant performance penalty. Some of our experiments therefore implement a relaxed variant of the above, which simultaneously performs seek reduction. At decision points, the scheduler picks any request within a threshold of the minimum virtual clock of all containers, which has the minimum positioning time from the current head position. This is somewhat similar to the Plso disk scheduler implemented in [VGR98].

2.2 Terminology

1. **Disk requests** are requests for disk I/O, issued to the disk subsystem. Disk requests are defined at a layer of abstraction below all the usual caching mechanisms,
so a disk request is guaranteed to reach the disk hardware and not be serviced from memory. These requests can initiate read or write I/O for an arbitrary number of bytes, up to a predefined maximum transfer size.

2. A disk scheduler is an operating system artifact that schedules disk requests for service on the available disk(s). For the purposes of this thesis, we restrict ourselves to a single physical disk.

3. A request issuer is a convenient executing abstraction whose code is directly responsible for initiating disk requests. There is some inherent ambiguity in the choice of a request issuer for various disk request types; we resolve this by identifying the highest-level executing entity that led to the immediate issue of the disk request.

- When an application issues a series of read() calls, some of which are uncached and cause disk activity, we hold the application responsible for issuing those requests. Similarly, if an application memory maps a file and accesses its pages, thus causing page faults, we hold the application rather than the VM subsystem responsible for generating the immediate disk requests. Further, we may talk of a process or a thread or some other operating system defined resource principal like resource containers assuming the role of the request issuer.

- An application could memory map a file and, without reading its contents from disk, directly write() out the memory region to a socket. In this case, page faults originate from the process running in kernel mode, and as such, we deem the I/O subsystem in the kernel responsible for issuing requests. The sendfile() system call behaves similarly.

- If an application performs file I/O, it may be required to first read some associated metadata, like directories and inodes. The application has no explicit knowledge of these I/O requests, so we choose to consider the filesystem as the request issuer.
Normal file data written to disk by applications generally gets buffered for some time, until the syncer daemon or equivalent wakes up and explicitly flushes them to disk. Writes to swap space are typically initiated by the pageout daemon. This is really a simplification, but that's the general idea.

4. **Idleness (of a request issuer)** With respect to disk activity, a request issuer is *idle* at some time instant if it does not issue any disk requests, *even when given a reasonable opportunity to do so.*

5. **Deceptive idleness** is a condition wherein a disk scheduler, at certain critical moments, incorrectly assumes upon the idleness of a request issuer.

2.3 The underlying problem: deceptive idleness

Disk schedulers typically judge a request issuer to be idle simply because it has no requests pending at decision points. From the application's perspective though, and from that of the system as a whole, this may be an improper criterion. For example, it is possible that the request issuer was continuously attempting to issue disk requests, yet did not get the chance to do so before decision points.

The disk scheduler always reorders the *available* requests according to the scheduling policy. The aggregate disk queue may be long and bursty on loaded systems [SCO90], but there may be no request at the decision point that enables the scheduling policy to make a good decision. Disk schedulers are generally work-conserving, and proceed with whatever requests are available. Naturally, application-level objectives of performance and quality of service are not met.

The rest of this thesis characterizes deceptive idleness and its impact on various disk schedulers, and finally provides a systems solution to address its various manifestations.
Chapter 3

Sources of deceptive idleness

The immediate question, therefore, is of why a disk scheduler might possibly develop an incorrect understanding of request issuer idleness. This section presents a categorized picture of these sources of deceptive idleness, without worrying about any of their consequences. The underlying message here is that many of these are fairly commonplace situations brought about by operating system design choices and standard programming practices.

We identify two fundamentally different reasons for deceptive idleness: request dependencies, and contention for some other resource. In practice, each of these could manifest in many different ways, potentially originating at the application level or from within the kernel.

3.1 Request dependencies

Imagine a request issuer generating a single, continuous stream of back-to-back disk requests. A few hundred microseconds after one request completes and the issuer gets control, it issues the next one. So from the application’s point of view, this request issuer is never idle.

Consider the same scenario from a scheduler’s viewpoint. A work-conserving scheduler is compelled to issue the next request (if available) immediately after completion of the previous one. At this decision point, control has not yet been returned to the issuer of the previous request. This issuer is waiting for the outcome of the previous request before issuing the subsequent one, and is unable to do so. Hence there are no requests pending for this issuer, leading the scheduler to falsely assume
upon its idleness. This is an example of how synchronous request generation can lead to a misunderstanding of request issuer idleness.

Not surprisingly, this effect manifests itself in practice in various different ways. If an application implements a single loop in which it invokes the `read(0)` system call on random disk blocks in a file, then we notice a chain of dependent requests at the application-kernel boundary. Upon completion of one request, control needs to be returned to userspace for the issuer to generate the next request.

In fact, applications running in user mode are not the only sources of dependent requests. Certain operations like `write` from file to network, and the `sendfile` system call have the I/O subsystem as the request issuer. These can potentially generate a stream of dependent read requests from within the kernel. A context switch to the application is not required for generating the subsequent request, yet we observe deceptive idleness due to the work-conserving scheduler acting immediately.

It may be argued that some of the above can be solved by suitably rewriting the application to use asynchronous I/O. However, this is not always true: sometimes request dependencies are unavoidably inherent in the very nature of these disk requests. This is the case with many common applications; for example, the apache webserver does not know of a future disk request before receiving the request. Some database index traversal operations may necessarily have to be performed one after another.

Such conditions may also naturally arise within the operating system. For instance, reading a file typically involves reading some associated metadata first. This could potentially involve several disk reads for inodes, indirect blocks and directory contents, which need to be performed one after another. As a pathological case, consider a fairly long chain of symbolic links that need to be traversed to access a particular file. Further, imagine that each of these symbolic link files is buried deep within a directory hierarchy with no common path components between them. To access the target file, many dependent disk accesses would be required.

Performing multiple simultaneous I/O operations (instead of issuing asynchronous
I/O) may also prove insufficient for solving deceptive idleness. Imagine a multi-process webservers that reads many small files scattered over an intricate directory structure. Each process incurs metadata dependencies, but the server as a whole would seem to issue enough requests whenever required. However, this need not always be the case: if many of these accessed files have a significant number of their path components in common, then they implicitly share the metadata associated with those directories. As a result, reading even different files may entail reading the same metadata block, thus generating a single stream of dependent requests for the entire server.

3.2 Resource contention

Even if request dependencies are altogether avoided, it may be possible to encounter deceptive idleness for quite different reasons. For example, there may be factors within the kernel that prevent an application from making progress at certain crucial moments. Such internal blockages could be a direct consequence of the disk scheduler. Despite the application trying to reach the point where it can issue requests, the scheduler will observe no requests actually being issued, and assume the issuer to be idle.

A situation like this could arise in practice due to contention for the main memory resource. A process that allocates and uses enough memory will eventually be forced to pageout some of it to swap. When this happens, another process that needs memory before issuing a disk request gets blocked, waiting for the first to finish paging itself out. Thus from the application’s point of view, there is no reason why it should not continuously issue disk requests. However, the scheduler momentarily sees no disk requests emanating from the second process, thereby considering it idle and usually takes incorrect action. This effect somewhat resembles priority inversion, where a high-priority process contends for a kernel resource that is currently held by a low-priority process.
3.3 Experimental characterization of deceptive idleness

We characterize all these sources of deceptive idleness, by performing a simple experiment. A Stride scheduler is assigned 1:2 proportions to two resource containers. These containers thus constitute the request issuers for this scheduler. One or two processes are bound to each container, as specified later. These processes continuously perform various actions as described, and never remain idle. The experiment involves measuring at every decision point, the number of request issuers (0, 1 or 2) that are busy according to the scheduler (i.e. have pending requests at those moments). These are averaged over periods of one second, so we could get $n_{busy}$ values continuously between zero and two. Ideally we want this number to be always equal to two, so as to reflect the application’s notion of non-idleness.

3.3.1 No deceptive idleness

We bind two processes to each resource container, and arrange for each of these processes to continuously issue randomly positioned disk requests. So at every decision point, at least one of the two processes in each resource container always has a request pending. Thus the number of busy resource containers ($n_{busy}$) is practically always 2, in figure 3.1.

![Figure 3.1: No deceptive idleness](image)
3.3.2 Explicit request dependencies

This experiment binds just one such process to each resource container. As soon as a request from either container finishes being serviced, the process bound to that container has not yet issued another request, and thus appears idle to the scheduler. We observe the effect of application-originating request dependencies, and \textit{nbusy} is almost continuously near 1 (figure 3.2). The scheduler is thus never able to successively service multiple requests from any container.

![Graph showing deceptive idleness due to explicit request dependencies](image)

Figure 3.2 : Deceptive idleness due to explicit request dependencies

We get an identical graph from request dependencies originating within the kernel. In this case, we map files into memory, and without actually accessing the contents, write them out onto a network socket. This is how many webservers (e.g. apache) normally operate. The kernel copies this memory region into network buffers, and incurs a stream of page faults. Handling these involves issuing a synchronous stream of 64 KB disk requests. The kernel could inform the scheduler that these requests are actually part of one continuous stream, but FreeBSD does not do that. Between requests, the scheduler incorrectly assumes that the issuer (I/O subsystem) has become momentarily idle.
3.3.3 Consequences of metadata sharing

The following experiment illustrates a slightly extreme case of sharing metadata between request issuers. We create a repository of small files, embedded in a four-level directory hierarchy. We allocate two resource containers, and bind two processes to each. These four processes read these small files in order, making sure they don’t read a file twice. Thus, they are implicitly forced to read the same metadata most of the time, and these have inherent dependencies in them. We therefore encounter short bursts of dependent metadata requests followed by short bursts of independent file data requests. These average to about 1.5 busy resource containers at any time. In the first few seconds, some higher level directories also needs to be read, so \( nb_{\text{busy}} \) drops even further in that period (figure 3.3).

![Figure 3.3](image)

Figure 3.3: Deceptive idleness due to shared metadata

3.3.4 Resource contention for memory

Resource contention for memory could lead to deceptive idleness in some pathological circumstances. In this experiment, two processes bound to each of the two resource containers issue random disk requests. From the application’s point of view, these requests keep each resource container always busy.
At the same time, each process allocates and consumes up to five times that much memory, and doesn’t free them. This can be viewed, for example, as memory used to store various processed versions of the data read from disk. When available physical memory in the system gets exhausted (about halfway into the experiment), the pageout daemon gets activated and starts writing to swap space. We associate these write requests with the appropriate resource container, depending on who owns the memory that is being written out.

While choosing pages for eviction, the kernel does not distinguish between processes that own these pages; it simply runs a global clock algorithm. It may therefore happen that one process gets blocked on memory allocation, and can proceed only after a process from the other resource container has written large chunks of memory out to disk. This leads to deceptive idleness, since the scheduler assumes the first process to be idle in this period. Thus in figure 3.4, nbusy begins with the correct value of 2, then drops down to 1 when severe pageouts induce deceptive idleness.

![Graph](image)

**Figure 3.4**: Deceptive idleness due to memory contention

None of the processes free any memory, and the system is thrashing heavily throughout the experiment. Soon after the depicted 60 seconds, the system runs out of swap space, becomes quite unresponsive, and kills the application. So this condition is a corner case: we shouldn’t be in such extreme memory shortage at all.
Chapter 4

Effects of deceptive idleness

We have seen how situations involving deceptive idleness can arise for various reasons. However, the question remains as to how it actually affects the behaviour of different disk schedulers. In this chapter, we examine how various manifestations of deceptive idleness cause fundamentally different kinds of problems with disk schedulers. In all forthcoming examples, the scheduler is forced to multiplex between request streams from different processes, at every decision point.

4.1 Seek-reducing disk schedulers

Consider an operating system equipped with any seek reducing scheduler, like Shortest positioning-time first (SPTF) or C-LOOK [WGP94]. These generally improve throughput by reordering available requests to minimize expensive head repositioning operations. Imagine an application base consisting of two processes $p$ and $q$, each issuing sequentially positioned 64 KB requests in different parts of the disk (as depicted in figure 4.1).

In one case, assume that both processes have initially issued all their requests. At almost every decision point, there are suitable requests pending for both processes. In the interest of seek reduction, the scheduler then services several requests for $p$, moves the head across the disk, and services several requests for $q$. This yields a throughput of 21 MB/s on our disk.

Instead, suppose both process issue a single synchronous stream of disk requests each, i.e. they issue a request only after the previous one has completed. The first request from $p$ is serviced, after which the work-conserving scheduler does not wait for
Figure 4.1: Deceptive Idleness on a seek reducing scheduler: (a) effective seek reduction, and (b) thrashing due to deceptive idleness

$p$ to generate its subsequent request. It assumes $p$ to have become idle, seeks across the disk to the pending request issued by $q$, and services that instead. A few hundred microseconds after this happens, the next request from $p$ arrives. It is now too late to service $p$’s nearby followup request, since disk scheduling is non-preemptible. The scheduler is thus forced to alternate between servicing requests issued by the two processes, thereby degenerating to FCFS and achieving a throughput of just 5 MB/s (due to 9ms of average positioning time for every 3ms of data transfer time for 64 KB).

If we had more than two processes issuing requests, then C-LOOK would behave like a round-robin scheduler, servicing just the one available request from each process before moving on to the next. On the other hand, SPTF would converge to the two closest requests on disk and alternate between them, starving the third process. Deceptive idleness does not always cause the scheduler to degenerate to a FCFS scheduler; the effect is often more subtle.

If the disk scheduler receives just one stream of sequentially positioned requests, then despite deceptive idleness, there would be no reason to reposition the head between requests. Any scheduler (even FCFS) would yield high throughput. However, with concurrent, synchronous request streams, we would experience large head seeks
for every request serviced - regardless of the seek reduction policy. This is the essence of how deceptive idleness causes seek-reducing schedulers to yield bad performance.

4.2 Proportional-share disk schedulers

Deceptive idleness can affect schedulers in ways other than just degrading performance. Consider a proportional-share scheduler like Stride [WW95] (with or without simultaneous seek reduction), assigned two resource containers A and B and assigned shares of 2:1. Assume disk-intensive applications are bound to these containers. These applications would thus expect the service that a busy request issuer deserves of the proportional-share scheduler. Assume for purposes of discussion that all requests target random blocks and are equally large, thus taking equal amounts of time to service.

![Diagram of request scheduling]

Figure 4.2: Deceptive Idleness on a proportional-share scheduler: (a) effective 1:2 allocation, and (b) skewed 1:1 allocation due to deceptive idleness

A typical implementation of this scheduler may service a few requests for container A, and correspondingly, a larger number of requests for container B. However, if processes in these containers collectively maintain only one outstanding request at any time, then a work-conserving scheduler becomes incapable of meeting the desired
proportions. Like before, it becomes forced to alternately service requests from the
two containers, achieving a service ratio closer to 1:1 (see figure 4.2).

4.2.1 Experimental characterization on Stride

We now examine a set of experiments that illustrate the deviation incurred on the
stride scheduler (with two resource containers), corresponding to different sources of
deceptive idleness.

4.2.1.1 No deceptive idleness

Figure 4.3 depicts the desired behaviour of the scheduler, without the presence of
deceptive idleness. Proportions of 2:1 are assigned to the two resource containers A
and B, and each container is associated with two processes. These two processes pro-
duce enough requests between them to avoid any delusions of idleness. Consequently,
we observe container A always receiving 66% of disk service. For example, after 60
seconds into the experiment, the two containers have received 40 and 20 seconds of
service time on the disk respectively.

![Figure 4.3: Proportional-share scheduler delivering desired proportions](image)
4.2.1.2 Explicit request dependencies

With only one process bound to each resource container, we experience explicit request dependencies as shown in figure 4.4. The number of active resource containers is almost always seen to be just 1, and we achieve close to 1:1 service allocated to the two containers. Application-level requirements of proportional allocation are not satisfied.

![Figure 4.4: Deceptive idleness due to explicit request dependencies](image)

4.2.1.3 Consequences of metadata sharing

As we have seen earlier, sharing metadata gives rise to deceptive idleness. Interspersed with file I/O, this leads to an average of about 1.5 resource containers that seem busy to the scheduler at decision points. We thus experience similar proportions being achieved by the stride scheduler (figure 4.5).

4.2.1.4 Resource contention for memory

As we have seen earlier, a heavily thrashing system can contract deceptive idleness due to one process needing memory that another process is trying to write out to swap. Figure 4.6 demonstrates the behaviour of a proportional-share scheduler under such circumstances – proportions are slightly violated on different occasions, and the stride
scheduler tries to eventually compensate for such skew. However, the stride scheduler is expected to deliver accurate proportions on a fine timescale, so such temporary violation is not allowed. Many runs of such experiments produce markedly different graphs, due to unpredictable pageout daemon behaviour.

This class of problems, though not very pronounced in its effect, actually illustrates the much bigger problem of global resource management. How much CPU or memory resources should the system allocate to a particular process, in order to enable it to issue its intended disk requests? How much CPU or memory blockage should a process tolerate before being declared really idle? These and other questions demand a deeper
analysis and greater solutions that we don’t attempt in this thesis.

One tentative approach towards solution for memory contention causing deceptive idleness, is that of self-paging as implemented in the Nemesis [Han99] operating system. When each process needs memory, the system writes out pages to swap from its own address space. Thus, blockage of a process for pageout I/O would affect the issue of its own requests only, and the problem of deceptive idleness should at least partially get solved. We have not explored this approach any further in this thesis.

### 4.2.2 More than two resource containers

Some of the above situations may look as though the scheduler has degenerated into a FCFS scheduler, due to complete lack of choice at every decision point. However, this is not always true; when scheduling from more than two resource containers, the effect of deceptive idleness can depend on specific idiosyncrasies of the scheduling algorithm. We examine this behaviour for two interesting and fundamentally different kinds of proportional-share schedulers.

#### 4.2.2.1 Stride, virtual clock based

Assume all requests take approximately the same amount of time to service. Consider three active resource containers $A, B$ and $C$, assigned proportions of 1:2:3 respectively. Deceptive idleness prevents two requests from any container being serviced consecutively. However, this restriction still allows the scheduler to meet the desired proportions. The schedule $C, A, C, B, C, B,$ etc. (with any permutations of $A$ and $B$'s) suffices for this purpose. Moreover, this schedule will actually be achieved by Stride, since virtual clocks “remember” deserved service and compensate if possible.

In contrast, consider proportions of 1:1:3 assigned to the containers. The constraint imposed by deceptive idleness now becomes too restrictive to meet these proportions. Requests are scheduled from $C, A, C, B,$ etc. and the skewed proportions of 1:1:2 are achieved instead.
Generalizing from these two examples, the Stride scheduler is capable of meeting proportions for a set of resource containers, if and only if the share on any one container does not exceed the sum of shares on the remaining containers. This condition is similar to the triangle inequality extended to polygons, where the length of a side never exceeds the sum of lengths of the remaining sides.

4.2.2.2 Lottery, randomized scheduling

A lottery scheduler [WW94] is a randomized proportional-share scheduling algorithm that operates by selecting a request from all runnable resource containers, with probabilities proportional to their assigned shares. On a somewhat coarse timescale, it intends to regulate the relative resource consumption rates of disk-intensive processes according to the assigned proportions. Though more meaningful for CPU scheduling, we analyze the effect of deceptive idleness on a lottery disk scheduler.

![Percentage disk service received](image)

**Figure 4.7**: Lottery scheduler behaving as desired

Let proportions of $\alpha_i$ be assigned to resource containers $A_i$, with sum of shares $S = \alpha_1 + \alpha_2 + \cdots + \alpha_n$. Let the scheduler actually achieve proportions of $\omega_i$, averaged over reasonable periods of time. Ideally we would expect $\omega_i = \alpha_i$, thus justifying correct behaviour. This does happen when all applications provide enough requests to the disk scheduler at the critical decision points. Figure 4.7 depicts an experiment
wherein three resource containers are assigned proportions of \( \alpha_{1..3} = 1:2:3 \) (i.e. 17% : 33% : 50%), and each has two processes bound to it, where each process issues randomly positioned back-to-back requests. Allowing for vagaries imposed by the inherent randomness, we observe \( \omega_i = \alpha_i \).

However, this stops being true if deceptive idleness is caused by having only one process bound to each resource container. After each request completes, explicit request dependencies in request generation result in the scheduler falsely assuming that the container that generated the request has momentarily become idle. Consequently the scheduler achieves the distorted proportions of \( \omega_{1..3} = 1:1.6:1.8 \) (i.e. 23% : 36% : 41%) instead of 1:2:3, as shown in figure 4.8.

![Figure 4.8: Lottery scheduler achieving skewed proportions due to deceptive idleness](image)

This strange behaviour deserves a mathematical explanation in the general case. With probability \( \omega_i \), a request from \( A_i \) gets serviced at every decision point. At the next decision point, this \( A_i \) has no pending requests due to deceptive idleness, and is therefore not chosen. So for \( j \neq i \), the conditional probability that a request from \( A_j \) gets chosen now is \( \frac{\alpha_j}{S - \alpha_i} \). Summing up these conditional probabilities, we get

\[
\omega_j = \sum_{i \neq j} \frac{\alpha_j \omega_i}{S - \alpha_i}
\]

or,

\[
\frac{\omega_j}{\alpha_j} + \frac{\omega_j}{S - \alpha_j} = \text{same for all } j
\]
Thus, equating for two different values of \( j \), we finally get

\[
\frac{\omega_i}{\omega_j} = \frac{\alpha_i (S - \alpha_i)}{\alpha_j (S - \alpha_j)} \quad \text{or} \quad \omega_i \propto \alpha_i (S - \alpha_i)
\] (4.1)

Thus, for assigned proportions of \( \alpha_{1,3} = 1:2:3 \), we get \( S = 6 \), which gives us \( \omega_{1,3} = 5:8:9 = 1:1.6:1.8 \), thus tallying quite well with the above experiment. To show that achieved proportions can be very different from assigned ones, \( \alpha_{1,3} = 1:10:100 \) gives us \( \omega_{1,3} = 1:9.1:10 \). When we introduce a very large number of resource containers, \( S \) grows large, so achieved proportions tend to approach the corresponding assigned proportions. These have been verified through direct experiment.

Thus the impact of deceptive idleness on the lottery scheduler diminishes when we have many resource containers at any time. But even when this is the case, it is common in practice to observe only a small number of resource containers in a system that are actually busy at any time. Secondly, it may seem possible to synthesize a set of assigned proportions for which the achieved proportions are just what one wants; however, this would typically not be possible, since the scheduler becomes sensitive to the number of busy resource containers at any time. Workarounds of this kind are thus difficult, if at all possible.

### 4.2.3 Simultaneous seek reduction

Practical deployment of a proportional-share disk scheduler would typically incorporate seek reduction as described in section 2.1.3. Deceptive idleness may allow such a scheduler to deliver the desired proportions, but may prevent the seek reducing aspect from working as desired. This is because requests for sequential blocks are generated often not at the level of resource containers, but by some entity more fine-grained, like a process or a thread. When this happens, binding multiple processes to each container may prevent deceptive idleness at the level of a resource container, but may not be enough to prevent deceptive idleness for each process. Proportional-share resource allocation typically cares about the former, and seek reduction, if any, is dependent on the latter.
Chapter 5

Workarounds for deceptive idleness

We have seen how a variety of fairly commonplace scenarios can easily develop deceptive idleness, and how this can result in most disk schedulers behaving in an undesirable manner. On the other hand, we ideally want to be able to plug any disk scheduler into an operating system and expect it to consistently function as intended. Thus we feel the need for solutions that address deceptive idleness.

We immediately recognize a possible workaround, based on *asynchronous prefetch*. If every active request issuer somehow proactively maintains even a single outstanding request of the right kind at decision points, then deceptive idleness can be prevented completely. The scheduler will have the option of choosing requests from any of the candidate issuers, including the one that issued the last request (this process is no longer mistaken to be idle). Preventing deceptive idleness thus requires either the application or the kernel to issue one or more supplementary requests before the current request has finished; these generally assume the form of various forms of asynchronous prefetch.

5.1 Application-driven prefetch

Applications can embrace programming paradigms and techniques that prevent the onset of deceptive idleness. We discuss several approaches, in each of which the application takes care to issue multiple outstanding requests of the right kind. By *kind*, we mean contiguous or proximal requests for seek reducing schedulers, without which there is little point in avoiding deceptive idleness.
5.1.1 Multi-process architecture

Perhaps the most intuitive approach is that of designing the application with multiple processes or kernel threads. If the application collectively creates enough disk activity, then we would expect several of these processes or threads to produce outstanding requests. Indeed, web servers like Apache and Flash are based on such a programming model. If we desire proportional disk service allocation between two Apache servers, then the scheduler always has enough requests to take sensible action.

Naturally, it would be difficult to expect all applications to be amenable to a multi-process or multi-threaded model. For example, a single process architecture with a sequential programming style may be perfectly suited to a movie player application.

There is also a performance problem with the above: requests issued by multiple processes often correspond to independent activities, and possess little spatial locality. For a seek reducing scheduler, these requests are not of the right kind. Though deceptive idleness is solved at the level of the entire application, it is still present in individual processes, which in this case matter for seek reduction. This results in a naive multi-process application (like Apache) achieving low disk throughput due to seeks between requests from its processes. An exception to this is when processes explicitly coordinate to issue a synchronous stream of requests, like the implementation of asynchronous I/O in FreeBSD.

5.1.2 Asynchronous I/O

If an application is aware of its future request issue pattern, it could proactively issue asynchronous prefetch requests. Alternatively, it can roll its own asynchronous I/O implementation using multiple processes or kernel threads. This method could be powerful and accurate, since the application is generally in the best position to estimate its future request pattern. Since the application controls the request being issued, we do not have the performance problem usually incurred by a multi-process architecture.
This section demonstrates how the \texttt{aio\_read()} system call can be effectively used to construct two parallel streams of requests, thus providing multiple outstanding requests at every instant. The semantics of \texttt{aio\_read()} are to issue a disk request and return immediately. A pre-specified signal is delivered to the process when the request finishes. Thus, issuing another \texttt{aio\_read()} at the end of this signal handler will result in a single chain of requests. However, here we need two streams of independent disk requests to prevent deceptive idleness. We therefore set up two disjoint copies of this request chaining mechanism, each with its own signal handler (\texttt{SIGUSR1} and \texttt{SIGUSR2}). These handlers need mutual exclusion between themselves, in order to select the next request consistently. This is easily achieved by masking each signal from the other while they are being handled, using \texttt{sigaction()} to set the signal mask.

There are several drawbacks to using asynchronous I/O. Firstly, asynchronous prefetch may not be possible even with complete application support, when the forthcoming request is intrinsically dependent on the currently executing one. For example, filesystem metadata or database index traversals, future requests for web servers, etc. Such a programming model is quite cumbersome to adopt, and obviously needs most applications to be rewritten for this purpose. Moreover, the implementation of asynchronous I/O in FreeBSD is still quite unstable (\texttt{aio\_read} is part of an optional POSIX realtime extension), and is not enabled in the default kernel configuration. Issuing explicit read requests using Unix API functions (instead of memory mapping the file) may entail more data copying and cache polluting, which could become expensive for in-memory workloads [PDZ99].

Thus, though asynchronous I/O provides us a powerful application-based workaround to address deceptive idleness, it may not be sufficient or desirable.
5.1.3 An mmap/mincore/read interface

The I/O subsystem in FreeBSD issues asynchronous prefetch requests for sufficiently large and sequentially accessed files. However, the VM subsystem does not do so on page faults (more on this in section 5.2.4). It is possible to work around this problem. The Flash webserver does this, by issuing `read()` requests explicitly on a memory mapped region. However, the `read` system call has performance implications when reading cached data, due to data copying and multiple buffering overheads [PDZ99]. Flash thus uses the `mincore()` system call on a memory-mapped file, and issues `read` requests only if the data needs to be actually read from disk. The overhead of copying is somewhat small compared to disk access time. In short, applications that desire good performance on current systems are sometimes forced to implement system-specific hacks.

5.1.4 Metadata caching

Metadata (directories, inodes, indirect blocks) is usually scattered quite randomly on disk, so there are few seek reduction opportunities involved. So the issue becomes that of transient violation of proportions or transient missing of deadlines, if metadata gets automatically shared in undesirable ways. This especially happens at the start of application execution, when cache misses in memory are mostly compulsory misses. Many applications are insensitive to such transient effects.

But hard real-time applications, for example, cannot allow for missing even a single deadline. The application-level workaround here is to periodically access the files involved, thereby bringing all metadata information to cache. Future metadata accesses therefore never actually reach the disk.

In FreeBSD, the metadata caching mechanism maintains an implicit cache size threshold of a few thousand entries. Caching more entries causes eviction of old entries. This may be undesirable for some large repositories of files (e.g., the CS department webserver stores more files). FreeBSD provides an indirect mechanism to
cache more directories, by enabling a sysctl named `vfs.vmiodirenable`. This makes the kernel view directories differently, and effectively increases the directory cache size (making it compete with the file cache).

Such situations are a rare occurrence, and such workarounds are clumsy. In general, it is often possible to eliminate deceptive idleness completely by taking proper precautions from the application. However, some of these are quite unnatural, and preferably avoided.

Secondly, modifying or rewriting applications may not even be allowed under many circumstances. A traditional general-purpose operating system may want to consistently provide seek-reduced best-effort service to existing applications. In some modern operating systems, resource management frameworks like resource containers [BDM99] and reservation domains [BGÖS98] allow for unmodified applications to be bound to the resource containers, and be allocated the designated resources.

In this world, it becomes essential for the operating system to be able to eliminate deceptive idleness, irrespective of what some potentially malicious application may try to do. A busy application that is expected to receive a fraction of the available disk resources should, under no pretext, be externally prevented from doing so. This idea forms the basis for analyzing transparent in-kernel solutions that solve deceptive idleness.

### 5.2 Kernel-driven prefetch

Filesystems can (and most do) try to guess future request patterns for applications, often at the granularity of file descriptors. They issue separate asynchronous prefetch requests\(^1\) for them. The usual reason is to overlap computation with I/O [SSS99], but this prefetching prevents deceptive idleness too. The primary purpose of this section would be to understand the limited scope in which this can be successfully performed.

\(^1\)different from synchronous readahead, where requests are enlarged to 64 KB to amortize seek costs over larger reads.
in various situations. The underlying message is that asynchronous prefetch becomes infeasible to implement and function correctly in a number of circumstances, some of which are quite important for some disk scheduler types. We examine a progression of such reasons. In each of these, the penalties of misprediction are high, thus forcing the prefetch heuristic to be very conservative.

5.2.1 Async prefetch may be completely impossible

We have seen examples of how request dependencies can be intrinsic in the nature of these requests. Database index traversal, or similarly, metadata access involve dependent disk accesses that not even the application can predict in advance. So any form of prefetch, including those initiated by the application, becomes impossible to issue.

5.2.2 Async prefetch may be impossible from the OS

Some applications are aware of their disk access pattern in advance, but to the kernel, this could appear totally random. Intelligently issued asynchronous I/O from the application is capable of solving this problem, but transparently prefetching from the kernel is not possible. In this and the previous cases, it may be argued that the benefit gained out of seek reduction is usually quite small due to the random nature of disk access. However, there is a gray area between sequential and unoptimizably random requests, where the predictive power of most prefetching heuristics does not suffice. For example, consider two processes, each accessing every alternate 64 KB chunk in different large files (one file per process). Neither FreeBSD nor Linux possess prefetch detection algorithms sophisticated enough to issue suitable asynchronous prefetch requests for such nontrivial access patterns. Eliminating deceptive idleness could fetch a benefit of a factor of two in this case.

Also, proportional-share schedulers do not care about request placement: they are expected to work equally well for sequential as well as random disk access, and in the
latter case, we clearly need solutions more feasible than asynchronous prefetch.

5.2.3 Async prefetch may need large files to detect

The I/O subsystem of FreeBSD issues asynchronous prefetch requests for the `read` system call, whenever it detects sequential disk access. But this involves internal logic that maintains some state, and detects and reacts to several sequential disk accesses. Thus, accessing medium-sized files may be regarded as random, simply because the conservative prefetch has not yet kicked in. This can adversely affect the performance of applications like the `apache` webserver that generally access medium-sized files.

5.2.4 Async prefetch may not be implemented: practical difficulties

Despite the possibility of performing prefetch, some subsystems of some operating systems are constrained by fundamental design choices and limitations that prevent them from doing so.

Perhaps the most conspicuous example of this is the VM subsystem in FreeBSD (as of 5.0) not issuing any asynchronous prefetch requests for disk accesses initiated by page faults. A `read()` request supplies a request size along with the block number, whereas a page fault provides less information. Consequently, the logic that detects and issues prefetch becomes more complicated, and is just not implemented. To compare, Linux has started supporting prefetch pagefauls only recently. This has serious consequences: webservers map small files into memory and directly write them out to a socket. The kernel incurs multiple page faults on this data, and immediately encounters deceptive idleness.

The `sendfile()` system call in FreeBSD issues explicit back-to-back disk requests, without asynchronous prefetch. In this case, prefetch should be relatively straightforward to implement, and is just not done so yet.

Large directories in FreeBSD, despite having approximately the same on-disk structure as regular files, do not generate asynchronous prefetch on being accessed.
This is because `readdir()` uses low-level interfaces that do not trigger prefetch mechanisms. This is an example of how OS design choices hinder the issue of asynchronous prefetch.

### 5.2.4.1 Common examples of this problem

One manifestation of deceptive idleness, viz. low disk utilization in seek reducing schedulers due to page faults not issuing asynchronous prefetch is a somewhat well-known phenomenon in the application development community. This is primarily regarded as a specific failure of some operating systems to provide asynchronous prefetch for page faults; fixing them would solve these concerns.

To avoid a possible denial-of-service attack on IRIX systems, the apache webserver serves files smaller than 4 MB by mapping them into memory and writing them out onto the socket. Beyond 4 MB, it switches strategies to reading them directly into a local buffer using `fread()`, and writing these chunks out. Ftpd-6.00LS in FreeBSD-5.0 does something similar, except that it hardcodes the switching point (again arbitrarily) to 16 MB.

The tradeoffs are hairy in either direction. Using `fread()` is clearly beneficial if the kernel does not implement asynchronous prefetch for servicing page faults from disk. On the other hand, the `mmap` approach is strongly preferable for files served from cache, because of many reasons: it reduces data copying (and thus cache polluting) costs, and it doesn’t occupy space in the application’s footprint for receiving and perhaps caching the data [PDZ99]. These costs are somewhat tolerable in comparison to disk I/O.

Figure 5.1 illustrates an experiment that studies the performance improvement of CSCAN with and without asynchronous prefetch. The apache webserver is used unmodified except for changing the `MMAP_LIMIT` threshold, to get apache_read and apache_mmap. The workload consists of a repository of many equally sized files, which are requested back-to-back by two clients. This experiment exercises only the
disk subsystem, so copy avoidance and other benefits of `mmap` are not shown.

![Graph](image)

Figure 5.1: Impact of asynchronous prefetch on seek-reducing scheduler performance

For small files, we observe `apache_read` behaving almost as badly as `apache_mmap`. As the filesize increases, slight amortization of other overheads causes `apache_mmap` to improve marginally. In contrast, the performance of `apache_read` shoots up to 75% of the maximum possible disk bandwidth. This is because the latter capitalizes on seek reduction opportunities by issuing asynchronous prefetch requests. Apache, in its default configuration, would pay the performance penalty of `mmap` until filesize reaches 4MB, and follow the `read` curve thereafter.

Thus, the kernel implementing asynchronous prefetch on page faults would completely eliminate the need for this messy tradeoff with seek-reducing schedulers, when requests are actually sequential. Of course, in this situation, applications streaming large files may perhaps want the kernel to implement some scheduler other than CSCAN (like ASPTF or GSPTF), to avoid incurring huge maximum response times.

To bring out the deeper nature of deceptive idleness, we later evaluate our solution using some experiments that are not limited by the above weaknesses in FreeBSD's asynchronous prefetch implementation.
5.2.5 Summary of this section

The key point of this section is that asynchronous readahead implemented and always correctly functioning in every I/O processing mechanism in the kernel is sufficient for solving deceptive idleness. However, prefetch mechanisms demand an accurate prediction of future access, have huge misprediction penalties, and are either impossible or difficult to implement for a variety of reasons. We therefore propose an alternative solution based to deceptive idleness, based on a non-work-conserving scheduling framework.
Chapter 6

Non-work-conserving scheduler solution

We have examined a variety of situations where a naive implementation of some disk scheduling policies can fail to meet application expectations. We have seen how deceptive idleness affects real applications in practice, and causes significant losses of performance and QoS accuracy. We therefore need a simple, application-transparent, low-overhead solution to deceptive idleness that enables any scheduler to function as intended. This solution should be suitable for implementation in a general-purpose operating system, and should smoothly and efficiently complement workarounds like prefetching, initiated either from applications or from the filesystem. Lastly, in the more common case of not encountering deceptive idleness, the solution should have little or no impact on system performance. This section describes such a solution.

Deceptive idleness manifests due to a combination of three factors: (a) disk-intensive applications concurrently issuing synchronous streams of disk requests, (b) the intrinsic non-preemptible nature of disk requests, and (c) a work-conserving scheduler that doesn’t wait for the subsequent request to arrive. Our solution takes the intuitive approach of changing (c) to wrap the scheduling policy in a non-work-conserving framework. At decision points, the scheduler now potentially waits for a short period of time for additional requests. Applications that quickly generate their subsequent requests can do so before the scheduler takes its decision; deceptive idleness is thus avoided. The disk is kept idle during this waiting period, as is the characteristic of all non-work-conserving schedulers. However, this wastage is not necessarily detrimental. The remainder of this section explains how a careful development of this method actually improves throughput consistently, and adheres to quality of service
goals better.

The question remains as to whether and how long to wait at a given decision point. A naive approach may make the simplification of waiting for a fixed duration at every decision point, irrespective of the scheduling policy. This obviously gives us poor performance, so we wait selectively. In particular, the scheduler waits for the duration over which it expects the benefits of waiting to maximally outweigh the waiting cost.

However, precise interpretation of costs and benefits depends on the scheduling policy. A seek reducing scheduler may wish to wait for contiguous or proximal requests, whereas a proportional-share scheduler may prefer weighted fairness as its primary criterion. To allow for such flexibility, we modularize our solution into a scheduler-independent core NWCS waiting mechanism, and separate, scheduler-specific NWCS decision heuristics. The waiting mechanism forms a framework around both the scheduling policy and the heuristic. It implements the generic logic involved in waiting (with timeouts etc.), and invokes the two methods exported by the heuristics: evaluate, which makes informed decisions about whether and how long to wait at a decision point, and collect_stats, which accumulates statistics required for making those decisions. These heuristics are implemented separately for each scheduling policy, and have access to the internal data structures of the policy implementation (figure 6.1).

The remainder of this section details the core NWCS mechanism, then spells out two assumptions, and then describes separate NWCS heuristics for seek reducing and proportional-share schedulers. Finally, it covers some implementation issues.

6.1 The NWCS waiting mechanism

A traditional work-conserving scheduler operates between two states, IDLE and BUSY, with transitions on scheduling and completion of a request. Applications can issue requests at any time; these are enqueued into the scheduler's pool of requests. If
the disk is idle at this moment, or whenever another request completes, a request is chosen from this pool and scheduled. This calls the scheduling policy function select, dequeues the chosen request from the scheduler pool, and dispatches it to the disk driver.

Our proposed NWCS mechanism forms a wrapper around a traditional scheduler. It invokes the scheduling policy to select a candidate request. However, instead of dequeuing it immediately, it first evaluates this request using the heuristic, which uses some nontrivial decision process to return either zero or a positive integer. Zero indicates that the heuristic has deemed it pointless to wait; we therefore proceed with the candidate request. However, a positive integer represents the waiting duration in microseconds that the heuristic has judged suitable. The scheduler initiates an event-notification based timeout for that period, and switches to a new WAIT state. Though the disk is inactive while in this state, it differs from IDLE by having pending requests and a ticking timeout.
Applications may issue new requests in this waiting period. In fact, a good heuristic would imply high likelihood that a new request will indeed arrive, and be preferable. This incoming request is not immediately serviced, nor is it even directly considered for evaluation. It is simply enqueued into the scheduler pool, and the policy selection routine is invoked to choose the best candidate request at this point. This approach retains the intrinsic properties of the policy like whether or not it has potential starvation, prevents duplication of scheduling policy code, and achieves an elegant separation of concerns between the policy and the heuristic. This candidate request is evaluated as before, and dispatched immediately (and the timeout is cancelled) if the heuristic decides thus. However, the waiting case is different: if the scheduler is in \textsc{wait} state and the heuristic decides to wait for an additional specified duration, then we disregard this duration and refuse to retrigger the timeout. This prevents unbounded waiting by repeatedly retriggering the timeout, since the suggested duration by the heuristic never exceeds a certain limit (15ms). If the timeout expires before any suitable incoming requests arrive, then we dequeue and dispatch the current candidate request. This algorithm is depicted in the state diagram in figure 6.2.

![State transition diagram for the NWCS waiting mechanism](image)

Figure 6.2: State transition diagram for the NWCS waiting mechanism
We cause up to milliseconds of delay between consecutive requests. The disk platter continuously spins under the head, so the target sector on the next request may be missed, and may require a complete 8ms rotation to return to. Thankfully almost all modern disks have a track buffer that prefetches and caches sectors as the disk spins by. This does not solve deceptive idleness, but it allows for an efficient NWCS solution.

6.2 Process-granularity assumptions

We tentatively make two simplifying assumptions about the granularity at which applications issue requests. We may observe a slight drop in performance if either assumption is violated.

6.2.1 Synchronous requests issued by processes

On completion of a request, we try to wait for the causally dependent request. This request could potentially originate from any process, leaving us very little information about how long to wait. However, for most applications, dependence between requests gets reflected in code structure. The subsequent request is typically generated by the same process, either in a loop or based on the outcome of the last request. It is rare that a group of processes explicitly coordinate to issue a synchronous stream of requests (an example of this is the FreeBSD implementation of aio.read using multiple kernel processes). This common-case application structure makes it sufficient to only wait for the process that issued the previous request. Therefore, assumption #1: Synchronous streams of disk requests are issued at the abstraction level of individual processes. These processes can be running in either user or kernel mode. If the system supports kernel threads, and if applications use them, then we should replace processes by kernel threads.

This assumption lets us add a clause to the NWCS mechanism that diminishes the impact of sudden variations in application behaviour, and corresponding heuristic
errors. Consider a process that generates a stream of rapid, good requests, and suddenly issues one bad request (due to filesystem layout, for example). The heuristic expects a good followup request to arrive soon, and decides that the policy-chosen candidate is bad enough to warrant further waiting. But the process has blocked on I/O, and will not issue any further requests. Assumption #1 allows us to disregard the heuristic decision, cancel the timeout and force an immediate dispatch of the chosen candidate.

6.2.2 Request homogeneity within a process

The previous assumption seems sufficient to accumulate sensible per-process statistics to assist in request evaluation. However, a process could be issuing two types of synchronous read requests, like accessing one file sequentially and another randomly. This would require statistics to be collected at the level of granularity of file descriptors, for example. We tentatively assume that this doesn’t happen, thereby stating assumption #2: Barring occasional deviations, there is reasonable homogeneity in the properties of synchronous requests issued by a process. Suggestions to relax both these assumptions are provided in section 6.8.

6.3 Seek reducing schedulers

This section describes the scheduler-specific NWCS heuristics for seek reducing schedulers such as SPTF, ASPTF and C-LOOK. The Shortest Positioning Time First [SCO90, JW91, WGP94] policy calculates the positioning time for each available request from the current head position, and chooses the one with the minimum. The goal is to design a heuristic that makes its waiting decisions to maximize expected throughput.

This heuristic evaluates a candidate request as chosen by the policy. The intuition is as follows: if this candidate request is close to the current head position, then there is little point in waiting for additional requests. Otherwise, using assumption #1, if
the process that issued the last request is likely to issue the next request soon (i.e.,
its expected median thinktime is small), and if that request is expected to be close
to the current head position, then we decide to wait for it. This waiting duration is
the expected 95-percentile thinktime, within which there is a 95% probability that
the request will arrive.

This idea can be generalized into a succinct cost-benefit equation. The key point
is to profitably balance the benefit of waiting, i.e., expected gains in positioning time,
against the cost of waiting, which is the additional time likely to be wasted. If \( \text{LP} \)
is the last request issuing process, and \( \text{elapsed} \) is the time since completion of the
previous request, then

| benefit     | \( (\text{calculate}\_\text{positioning}\_\text{time}(\text{Candidate}) - \text{LP.e}\_\text{expected}\_\text{positioning}\_\text{time}) \) |
| cost        | \( \max(0, \text{LP.e}\_\text{expected}\_\text{median}\_\text{thinktime} - \text{elapsed}) \) |
| duration    | \( \max(0, \text{LP.e}\_\text{expected}\_\text{95percentile}\_\text{thinktime} - \text{elapsed}) \) |
|             | \( \text{return (benefit > cost ? duration : 0) } \) |

Positioning time for the candidate request is calculated using a suitable estimator
(more on this in section 6.7). The three expected times are gleaned from online statis-
tics collected by monitoring newly arriving read requests. Specifically, the expected
positioning time for each process is a weighted average over time of the positioning
time for requests from that process, as measured upon request completion. The decay
factor is set to forget 95% of the old positioning time value after ten requests, so it
adapts fast. We could alternatively track the expected seek distance of a request from
the previous request issued by that process, and calculate expected positioning time
on the fly.

Expected median and 95%ile thinktimes are estimated by maintaining a decayed
frequency table of request thinktimes for each process. Thinktimes are computed
from the point of completion of the last request \( \text{for this process} \), to the current time.
If however, the scheduler already has a read request queued for this same process,
then we treat this new request as asynchronous and set its thinktime to zero. We maintain 30 per-process buckets that store the count of requests that arrive after various thinktimes, ranging from 0 to 15ms at a granularity of 500μs per bucket. We decay these bucket counts by reducing all of them to 90% of their original value for every incoming request for that process. The thinktime distribution usually looks like a bell curve. Assumption #2 guarantees that this curve has at most one hump; for many applications, it is located at about 1ms. Lastly, we calculate the median and 95th points of this curve. We do all the above for every incoming synchronous request.

For a policy as conceptually simple as SPTF, this is the perfect greedy heuristic: it always tries to make waiting judgments with throughput as the only goal.

6.3.1 Variant: ASPTF: Aged SPTF

SPTF suffers from poor starvation resistance, potentially allowing distant requests to starve. Wrapping it in the NWCS solution only increases this likelihood. Therefore a weighted variant of SPTF has been proposed, which explicitly bounds response time. Aged-SPTF [SCO90, JW91, WGP94] remembers the queued time for individual requests, and continuously raises the priority of old requests. A request with sufficiently high priority overrules the SPTF decision.

Firstly, the SPTF heuristic would work unmodified. When ASPTF chooses a distant request that is too old, the SPTF heuristic would be unaware of this. It may decide to wait for additional, nearby requests. However, even if any new request from the last process arrives in this period, the policy will continue to pick the same old request. The last process will get blocked, and we would proceed with the policy-chosen candidate. This has worked exactly as desired, except for one unnecessarily wasted thinktime on each of the infrequent occasions that this happens. This minor suboptimality can be fixed by making the heuristic aware of the policy internals. Whenever a request is chosen that doesn’t coincide with the corresponding SPTF
selection, then we decide not to wait.

6.3.2 Variant: C-LOOK: Cyclic look

One extremely popular scheduling policy is the Elevator algorithm or C-LOOK, implemented in most Unix-based operating systems. This policy makes the head move in a unidirectional manner, servicing all requests in one direction, and then starting all over at the beginning. Though this policy only makes moderate sense from the point of view of either throughput or response time, we attempt an NWCS heuristic.

We base this on SPTF, with one additional clause. The statistics collection module in the heuristic additionally maintains a decayed expectation of the seek direction: forward or backward. On evaluating a request, if the current candidate involves a forward seek and the expected next request has a fairly high likelihood (more than 80%) of a backward seek, then we bypass the cost-benefit equation and decide not to wait. In the opposite case, we wait for the usual amount of time. For applications performing random access, with roughly 50% of the seeks pointing in each direction, this heuristic for C-LOOK is not ideal. This is because C-LOOK itself is poorly suited for handling this case.

6.4 Proportional-share schedulers

We examine NWCS heuristics designed for proportional-share schedulers like Stride [WW95] and Yet-another Fair Queueing (YFQ) [BBG+99]. Stride maintains weighted virtual clocks to remember the amount of disk service received by each process. A request is chosen from the resource container with the smallest virtual clock, so as to advance them in tandem.

Unfortunately, deceptive idleness forces these virtual clocks to go out of sync. Some resource containers do not generate enough requests in time, and their virtual clock lags behind. Genuinely idle containers also lag behind, but their expected thinktimes are high. Our heuristic for Stride is therefore as simple as waiting for
requests from the process LP that issued the last request, if three conditions are met:

- the container bound to LP has no pending requests at the decision point,
- this container has a virtual clock smaller than the minimum virtual clock of all containers with available requests (min\text{clock}), and
- the process LP has an expected thinktime smaller than 3ms.

The 3ms threshold is arbitrary: there is no sensible way to balance weighted fairness against performance. 3ms is larger than the thinktimes for most applications, without being too large to overly degrade performance. As before, we wait for the 95\%ile point of the thinktime distribution for this process.

6.4.1 Variant: YFQ: Yet-another Fair Queueing

The Stride scheduler explicitly requires active processes to join the candidate pool, and those inactive for extended periods to leave. A joining process is assigned a virtual clock value equal to the global virtual clock at that time, which is effectively the minimum of virtual clocks of all active processes, if any. This “bumping up” of virtual clocks prevents an inactive process from accumulating virtual time and monopolizing the resource upon later activation.

In contrast, schedulers like Yet-another Fair Queueing (YFQ) and Start-time Fair Queueing (SFQ) [GGV96] deduce the notion of process idleness, by checking whether a queue is empty at the time a new request arrives. Deceptive idleness causes this condition to be insufficient. A heuristic for YFQ or SFQ thus needs to make waiting decisions, as described above for Stride. In addition, it needs to explicitly convey this information to the scheduling policy, which should use this as the idleness criterion for bumping up virtual clocks.
6.5 Heuristic combination

Consider a relaxed proportional-share scheduler that picks requests from containers with virtual clock between minclock and minclock + τ (where τ could be 100ms). Among these, it chooses the request with minimum positioning time [VGR98]. Such schedulers demand general methods of combining NWCS heuristics for two separate policies, e.g., proportional-share and SPTF.

A naive approach is to separately evaluate the candidate request on each of the two heuristics, and return the larger of the two return values. In other words, if the waiting decision is taken for either reason, then the combined heuristic will conservatively choose to wait.

There are two minor performance problems with this approach, based on the nontriviality of combination of the two policies. If say the SPTF heuristic decides to wait for a favourable request, but if it is known beforehand that the combined policy will definitely not choose this request, then there is little point in waiting. This can be solved by using more inside information from the policy. Secondly, the proportional-share scheduler has relaxed due to the introduction of τ; the naive heuristic has an overly stringent lagging-behind condition. The heuristic should check for potentially lagging behind, which happens when the container bound to the process that issued the last request has a virtual clock smaller than minclock + τ.

6.6 Real-time schedulers

We have not implemented the heuristic for any real-time scheduler, but we make an observation regarding the pure Earliest Deadline First (EDF) policy. At decision points, this policy chooses the available request with the smallest remaining time to deadline. It does not take seek effects into account, and is fused in practice with some seek optimizing scheduler.

Compare pure EDF with SPTF: at decision points, the latter calculates positioning times from the current head position to each available request, and chooses the
smallest. The heuristic for SPTF can thus be morphed to one for EDF by substitute positioning time by deadline. In SPTF, one hopes that the last request issuer produces a new request with small positioning time. Correspondingly in EDF, the last issuing process may spend some thinktime and then issue a request with a small deadline. If the sum of the leftover thinktime and the deadline of the new request is smaller than the deadline of the candidate request, then pure EDF justifies us in opting to wait.

A more practical real-time scheduler would include other factors like positioning time for seek reduction, and total request service time to see whether the current candidate request can be squeezed into the expected deadline of the forthcoming request.

6.7 Implementation notes

We implemented the NWCS framework and heuristics in the FreeBSD-4.0 kernel. The code comprises of a kernel module of about 1500 lines of C code, and a small patch to the kernel for necessary hooks into the scheduler and disk driver. Our experiments are run on a single 550MHz Pentium-III system, equipped with a 7200rpm IBM Deskstar 34GXP IDE disk and 128 MB of main memory. There are two practical difficulties: calculating positioning time for requests, and building an inexpensive event-based timeout mechanism.

Estimating access time for requests is nontrivial due to factors like rotational latency, track and cylinder skews, and features of modern disks like block remapping and recalibration. Nonetheless, much work has been done in this context [JW91, HC00, RW94], and it is possible to build a software predictor with over 90% accuracy. However, we used a much simpler block number based approximation to positioning time. We wrote a user-level program that performs some measurements (taking about 3 minutes) and fits a smooth curve through these points. This method automatically accounts for seek time, average rotational latency and track buffers. This has an
accuracy of only about 75%, but the NWCS heuristics are generally insensitive to this error.

There are many possible timer mechanisms to choose from. We use the i8254 Programmable Interval Timer (PIT) to generate interrupts every 500\(\mu\)s, and build a simple timeout system over that. Experiments demonstrate how this rather inaccurate timer is amply sufficient for our purposes. Each interrupt causes a processing overhead of about 4\(\mu\)s on our hardware [AD99], thus causing about 1% CPU overhead on computational workloads. Other timeout mechanisms can be used in place of the i8254, if higher accuracy is desired. Some pentium-class processors (mostly SMPs) have an on-chip APIC that delivers fine-grained interrupts with an overhead of only 1 to 2\(\mu\)s per interrupt. Alternatively, soft-timers [AD99] pose an extremely light-weight alternative.

### 6.7.1 Tagged queueing in SCSI controllers

The NWCS algorithm, as presented above, applies to IDE disks that have no additional queueing within the controller. It requires only one modification to work for SCSI disks with tagged queueing: the \texttt{busy} state variable needs to be extended to an integer that holds the number of outstanding requests. The algorithm automatically and correctly handles some subtle issues raised in this context, as follows.

Even when the disk is busy, \texttt{schedule()} gets invoked multiple (usually up to 4) times. The controller expects the scheduler to supply as many requests if available, and performs seek reduction (usually SPTF) among those internally queued requests. A scheduler that efficiently eliminates deceptive idleness should therefore do the same under normal circumstances, but effectively disable tagged queueing whenever deceptive idleness sets in. This prevents the disk controller from scheduling a request before the subsequent request from the last process has arrived.

The above heuristic (with the \texttt{busy} modification) does exactly this, because whenever the heuristic decides to wait, the scheduler pretends (to the controller) as though
there are no more requests available. Thus if a process does sequential disk access with asynchronous prefetch, both these requests will get scheduled to the controller, and no more (e.g. requests from other processes). This exploits tagged queueing just as much as is performance critical. If a process performs sequential disk access without asynchronous prefetch, then requests will get scheduled one after another — like IDE controllers. For processes with huge think times, or for random disk access, the scheduler never decides to wait. So tagged queueing gets exploited to the fullest.

There is a subtle problem with above reasoning, but it automatically gets resolved. The scheduler knows the set of requests that have been dispatched to the controller, and that one of them is currently receiving service. It doesn’t know which of those requests will be the last one to execute. It needs to know this for two reasons.

1. To perform SPTF accurately, i.e. to schedule the request closest to the last serviced request on disk. Luckily we implement the same SPTF policy both inside and outside the controller, so we expect that our positioning time estimate is accurate enough to reasonably reflect the internal SPTF. So the controller would mostly not reorder the dispatched requests, and this problem almost vanishes.

2. If requests dispatched to the controller belong to different processes, then our heuristic ideally needs to check if a new request arriving later is proximal to the request issued by any of those processes. However the heuristic only checks the last process. In fact, not checking for all processes is beneficial to the heuristic. There is a subtle redeeming reason for why this will also generally work. Suppose process $p$ issues requests $r_1$ and $r_2$ (where $r_2$ may be a prefetch request), and process $q$ issues $r_3$. Then the scheduler may dispatch both $r_1$ and $r_2$ to the controller. At this point, one of two things happen:

   ◦ if there is deceptive idleness, i.e. $p$ is likely to receive a subsequent good request soon, then we stop scheduling requests. The set of requests issued to the controller belongs to just one process, and we’re okay.
if there is no deceptive idleness for process $p$ (regardless of $q$), then the scheduler
goes ahead and dispatches request $r_3$ also to the controller. Implicitly our
heuristic assumes that $q$ has issued the request that is serviced last. This would
work favourably for our heuristic that only checks the last issued process, since
$q$ may potentially be eligible for waiting, and $p$ most likely won’t be.

6.8 Potential improvements

Our proposed heuristics collect statistics about application behaviour, assume them
to be valid, and use them to make decisions. We suggest two approaches to im-
proving this NWCS solution, in both of which the heuristic has a smaller chance of
getting misled. These are aside from the obvious ones of making the timeouts and
the positioning time estimator more accurate.

6.8.1 Accumulate more statistics

A few incorrect decisions are usually acceptable, but sustained errors are not. We
can therefore take a feedback-based approach to supplement or validate our heuristic
knowledge with additional measurements:

1. In addition to tracking expected thinktimes and positioning times, we could
   collect information about the variance of these estimates. This gives the heuristic
   an idea of how accurate these estimates really are. 2. We could keep track of how
   frequently timeouts expire for each process. If these exceed some threshold rate, then
   regardless of all other notions of accuracy, we know that something is wrong. 3.
   We may not be able to predict positioning times accurately, but we can measure it
   after the request has completed service. This tells us something about the error in
   the estimator, and thus, our confidence in the correctness of a future decision. 4.
   Some applications use aio_read to issue requests synchronously; we can determine this
   post-facto, and remember the fact for future decisions.
6.8.2 Fix the process-granularity assumptions

Some proportional-share schedulers work at a higher level of abstraction than processes, like resource containers [BDM99] and reservation domains [BGÖS98]. Sometimes a group of processes may collectively issue requests. Applications often simultaneously generate different access patterns on different file descriptors. Some programs may issue two kinds of disk requests from two different parts of the program code, but on the same file descriptor. Seek reduction intrinsically deals with requests in the same region on the disk.

The general solution to all this is to collect statistics at each level of abstraction, i.e. processes, threads, instruction pointer for thread, file descriptors, and disk region – along with their variances. The heuristic then chooses the highest consistent level out of these, which has low variance and is expected to be correct. More tricks can be played to reduce computational overhead of such a system.
Chapter 7

Experimental evaluation

This section evaluates the above NWCS solution on several microbenchmarks and real workloads. All experiments are run with and without the NWCS module enabled; our underlying claim is that this transparent kernel-level solution can eliminate deceptive idleness. With practically no overhead, it can achieve significant performance improvement and adherence to QoS objectives in a variety of cases, and almost never any degradation.

7.1 Microbenchmarks

Three sets of microbenchmarks serve to illuminate the workings of the NWCS solution in a range of circumstances. We look at variations in access patterns and thinktimes for seek reducing schedulers, followed by the behaviour of a seek-reducing proportional-share scheduler.

All seek reducing experiments below use the ASPTF scheduler, which performs shortest positioning-time first scheduling with a bounded delay of 1 second. This achieves performance to within 1% of SPTF, while preventing starvation. For proportional-share scheduling, we use Stride coupled with ASPTF, with a tighter fairness threshold of 100ms (smaller than the switching threshold of ASPTF).

We employ two metrics of application performance: the application-observed throughput (in MB/s), and the disk utilization. Disk utilization in a time interval is the fraction of time that the disk spends servicing requests. This includes head positioning time and data transfer time, and excludes idle time (i.e. utilization is not the fraction of time spent doing useful work).
7.1.1 Different access patterns

This experiment demonstrates performance improvements due to NWCS, with and without workarounds like asynchronous filesystem prefetch. Two processes rapidly issue 64 KB read requests into separate large files, using either `read` or `mmap`. These accesses are either sequential, or for every alternate 64 KB chunk, or randomly positioned within their respective files (figure 7.1).

![Graph showing throughput and disk utilization](image)

**Figure 7.1**: Different access patterns

Read induces kernel-driven asynchronous prefetch for sequential access, and hence achieves almost full disk bandwidth. File layout forces it to occasionally skip a block, and being conservative, prefetching becomes imperfect. The NWCS solution improves this by 5%, by steadily fetching blocks for one file until ASPTF forces it to switch. On the other hand, page faults in the mmap-ed region do not issue any asynchronous prefetch. NWCS achieves four times better throughput than before, while causing the disk to remain idle for about 5% of the time. This trend will be reflected later: moderate improvements for read, large ones for mmap.

Accessing alternate chunks defeats the FreeBSD prefetch heuristic. Both read and mmap achieve only 5 MB/s; in fact, read can go much lower for smaller chunks
(mmap cannot, because of synchronous prefetch). NWCS improves throughput to the maximum that can be achieved for alternate blocks, i.e. half the disk bandwidth. In the random access case, the minor (5% and 7%) improvements by NWCS is because each process is performing random access within its respective file, where some seek optimization is possible. These two lie in the gray area between sequential access (where kernel-level prefetch is usually possible) and unoptimizably random access.

### 7.1.2 Varying thinktimes

This set of four microbenchmarks illustrates the impact of waiting for applications that take different amounts of time to issue the subsequent request. Two processes map separate, large files into memory, and access these pages sequentially (thus avoiding kernel prefetch). For every 64 KB, they pause for some amount of time.

#### A. Symmetric processes

Consider figure 7.2, where time $t$ on the x-axis represents the duration in milliseconds that both processes spend waiting. For values of $t$ up to 8ms, the original system thrashes between requests from the two processes, fetching only 5 MB/s. At about 8ms, the waiting time becomes comparable to request service time, and utilization for the original system starts falling below 100%. Occasionally deceptive idleness gets avoided by servicing two successive requests for the same process. This fades away for larger values of $t$.

For NWCS: when $t = 0$, we see the familiar situation where throughput is four times that of the original system. For larger values of $t$ up to 8ms the effect of waiting becomes increasingly burdensome on throughput and utilization, and this huge improvement steadily reduces. At about 8ms, the waiting time becomes comparable to request service time, and the cost-benefit equation tips the other way. It starts approximating the original system to an increasing degree, until for very large thinktimes (i.e. on an almost idle system) it plays no role. Most applications have very
short thinktimes when busy, in the region of 200\mu s to 2ms. Hence NWCS is expected to fetch large benefits.

B. Asymmetric processes

Consider an alternative scenario in figure 7.3 where only one (slower) process waits for duration $t$. The original system thrashes for $t$ up to 12ms, but beyond that, two or more requests arrive for the quick request for every request from the slow one. This causes partial avoidance of deceptive idleness, due to which performance gradually improves for increasing $t$. With NWCS: throughput first decreases due to increased waiting for both processes, till a point (4ms) where it starts waiting for the quick process but not for the slow process. Throughput quickly rises back to the maximum, with requests from the slow process serviced only when ASPTF induces a switch. Note that ASPTF only guarantees non-starvation, not fine-grained fairness.

C. Random thinktimes

We wish to know how badly an application will perform if it defeated the thinktime heuristic. Interestingly, if a process waits for a random duration uniformly distributed
between 0 and \( t \), it performs almost as well as the deterministic counterpart. This is because the expected median thinktime is judged to be roughly \( t/2 \), and the expected 95th percentile thinktime becomes \( t \).

D. Adversary

So we wrote an intelligent adversary. Two processes wait for different durations as follows: they issue \( n \) rapid requests, then wait for a duration that just exceeds the timeout set by the kernel, and repeat. Graphs for varying \( n \) are depicted in figure 7.4. For \( n = 0 \), the NWCS solution can cope with all requests arriving slowly. But for \( n \) between 1 and 4, the NWCS heuristic performs slightly worse than the original system. This indicates that even in the most convoluted applications, degradation expected is minimal. Interestingly, an analogous situation arises in practice when applications issue very large read requests, and the kernel breaks them up into 128 KB chunks. We resolve that by having the filesystem inform the NWCS heuristic of this fact.

Many timeouts expire in this experiment, so it proves useful for judging our sensitivity to the accuracy of the timer. We changed the granularity so as to tick every 50\( \mu \)s, 200\( \mu \)s, 500\( \mu \)s and 1ms. There was less than 10% of difference in throughput
between these trials. This is also supported by a similar experiment on the Apache webserver, where the difference was negligible.

7.1.3 Proportional-share scheduling

The following microbenchmark demonstrates the functionality of a combination of the proportional-share and the seek-reducing heuristics. As before, two processes \( p \) and \( q \) sequentially access separate memory-mapped files rapidly. The underlying stride scheduler is assigned proportions of 1:2 for the two processes (with \( q \) getting the higher share). These proportions are in terms of utilization, not throughput.

Consider figure 7.5. In the original case, the scheduler multiplexes between requests from the two processes, and achieves 1:1 proportions with the low throughput of 5 MB/s. When we turn on the proportional-share heuristic, it realizes that process \( q \) is falling behind, and waits for it. Taking seeks into account (9ms seek time, 3ms data transfer time for every request), it achieves 1:2 proportions by servicing 6 requests from \( q \) for every request from \( p \). This can be calculated to yield slightly more than twice the throughput of the original case, and indeed, it does. It results in a 2% drop in total utilization, as is reflected by both allocations proportionally decreasing.
The above only waits for process $q$; when the seek reducing heuristic is also turned on, it realizes the optimization potential in waiting for both processes. This then services multiple requests from each process, thereby achieving 21 MB/s and dropping total utilization to 95%, while retaining proportions of 1:2.

If the requests were random, the proportional-share heuristic would still wait for process $q$, though the seek reducing one wouldn’t wait for either. This demonstrates how our combination heuristic works under all circumstances, eliminating the need for a real benchmark.

### 7.2 Real workloads

Solving deceptive idleness can clearly bring about significant benefits on microbenchmarks, but what is its impact on real applications? To see this, we use two real applications, and two standard benchmarks that are expected to mimic real scenarios.
7.2.1 The Andrew Benchmark

The Andrew Benchmark [HKM+88] understands filesystem behaviour on a typical fileservelike workload. It consists of each client performing five phases: (a) mkdir, which creates $n$ directories, (b) cp, which copies a repository of 71 C source files to each of these $n$ directories, (c) stat, which aggressively lists all directory contents, (d) scan, which reads all these files using grep and wc, and (e) gcc, which compiles and links them. We used $n = 500$ copies of the repository, which exceeds the main-memory cache. We configured this benchmark with two clients to simulate concurrent access on a fileserver, with each client acting on a separate instance of the $n$ repositories. This experiment uses the same ASPTF scheduler as before, with and without NWCS enabled.

![Figure 7.6: Andrew Benchmark](image)

A breakup of the execution times for individual benchmark phases is presented in figure 7.6 (The graph depicts the gcc bars scaled down by a factor of 3). Consider the scan phase, which is the only one that issues streams of synchronous read requests. NWCS transparently brings about a drop in execution time for this phase by 54%. Both grep and wc on FreeBSD use read, not mmap, and thus would have the advantage of kernel prefetch. However, individual files are small, so this prefetch plays
practically no role. Major seek optimization happens here due to the files being in the same directory, and thus closely positioned on disk. NWCS enables the scheduler to capitalize on these seek opportunities and halve the execution time.

Other disk-intensive phases improve by smaller amounts: 16% for mkdir with synchronous metadata writes, and 5% for cp and stat each (the latter actually gets cached in memory). The CPU-intensive gcc phase shows an execution time degradation of 1.7%, due to the i8254 timer and the NWCS heuristic execution overheads. This phase strongly dominates total execution time, so that the overall benchmark shows an improvement of 8.4%.

Performance with one client is the same with or without NWCS; indeed, when there is only one stream of synchronous requests, NWCS plays no role. Increasing the number of clients to 8 shows practically the same performance as two clients: the scan phase improves by 57% in this case. This confirms the applicability and scalability of NWCS to busy file servers.

7.2.2 The Apache webserver

The Apache web server [Apa] employs a multi-process architecture to service requests from clients. Requests that miss in the cache are serviced from disk by the respective process. This happens frequently for web servers with large working sets, to the point of being disk-bound. For files smaller than 4 MB, the web server mmaps the file and writes it out to socket; for larger files, it reads the data into application buffers (this was done to prevent some swap-based DoS attack on IRIX systems). Many other web servers and ftp servers use similar mechanisms for file transfer.

We configure this webserver to always use either read or mmap, and present results separately for both. We ran the Apache web server with 3 client machines and 16 client processes each; this should generate enough concurrency that is typical of web servers. We tried varying the number of clients, but it made little difference to performance. These clients rapidly issue random requests for distinct files. The files themselves are
chosen from a real workload corresponding to the CS department webserver at Rice University. These files have an average size of 34 KB. The file distribution is such that 95% of these files occupy 238 MB, whereas 99% occupy 814 MB. The scheduler, as before, is ASPTF.

![Graphs showing throughput and disk utilization](image)

**Figure 7.7**: The Apache webserver

Figure 7.7 characterizes the observed throughputs and utilizations. As before, we observe a large 56% improvement in throughput for mmap, but only about 16% for read. Unlike in the Andrew Benchmark, all clients to apache generate random requests to the entire repository, so requests to an individual apache process do not possess any special locality across files. So seek reduction opportunities are mainly in terms of servicing each file fully before moving on to the next. Many files are too small for any optimization. The intermediate-sized files are candidates, but conservative filesystem prefetch does not occur. NWCS makes the 16% difference in this region. Prefetch occurs for reads on large files, but not for mmap. This accounts for the large difference in performance between the two methods of access. In the default configuration (with mmap or read depending on file size), apache performs closer to the read case, indicating how files larger than 4 MB dominate performance.
### 7.2.3 The GnuLD linker

This experiment involves the last stage of a FreeBSD kernel build, starting from a cold filesystem cache. The GNU linker reads 385 object files from disk. 75% of these files are under 10 KB, whereas 96% are under 25 KB. After reading all their ELF headers, GnuLD performs up to 9 (usually about 6) small, non-sequential reads in each file, corresponding to each ELF section. These reads are separated by computation required for the linking process.

![Figure 7.8: The GNU Linker](image)

The experiment in figure 7.8 demonstrates the performance of one and two simultaneous instances of GnuLD on disjoint repositories. We use two schedulers this time, ASPTF and C-LOOK. With one synchronous request issuer process, both schedulers result in execution times of about 1.8 seconds each. We would expect this to double with two instances of GnuLD. However, deceptive idleness (and no prefetching) causes an execution time rise by a factor of 5.5 instead.

NWCS brings about a benefit of 68% in the ASPTF case, and causes performance to scale almost exactly as needed. The C-LOOK scheduler, on the other hand, always insists on unidirectional movement. Since the object files are accessed in arbitrary order, C-LOOK does not allow NWCS to optimize fully. Hence we achieve a perfor-
mance improvement of only 48%, and an execution time 56% higher than the ASPTF case.

7.2.4 The TPC-B database benchmark

The TPC-B benchmark [Cou94], proposed by the Transaction Processing Council in 1994, exercises a database system on simple, update-intensive transactions. Though it is declared to be outdated, it serves to illustrate the impact of NWCS on a read-write workload. The essential idea is to have clients making random update queries into a large database. We use two clients in our experiments. Individual records in the database are expected to be at least 100 bytes large. Unfortunately, the database system (MySQL) has computational overheads that make it CPU-bound for record sizes of 100 bytes, or even 1 KB. We therefore use 64 KB records, and a database size that exceeds main memory size.

![Graph showing performance comparison]

Figure 7.9: The TPC-B database benchmark

Figure 7.9 shows four experiments. The first one has all queries directed towards one database, so seek optimization opportunities are few. The second bar has requests
targetted from the two clients to *two separate* database files. This leads to choices of seeking within and between the files. The third and fourth experiments are similar, but the update operation is replaced by just a select.

An update query reads the record first, and then issues a delayed write. The presence of enough delayed writes can give the scheduler more choices, and reduce the effect of deceptive idleness. This happens in the first experiment, where the net improvement is just 4%. When we have two databases that are physically separated on disk, the impact of NWCS is better seen, as in the 28% improvement. Finally, the NWCS advantage is best brought out without any delayed writes, i.e. when the update operation is reduced to just a select. We observe throughput improvements by 17% and 60% for the same and different databases respectively.
Chapter 8

Related work

This chapter examines three broad categories of related work. We overview various scheduling algorithms, then move on to techniques that have been proposed to improve disk request handling in the I/O subsystem (external to the scheduler). Finally, we look at scheduling concepts relevant to other domains like CPU scheduling and rate-based flow control.

8.1 Scheduling algorithms

The last thirty years have seen an enormous amount of research in the area of disk scheduling algorithms [SCO90, WGP94, BBG+99, HC00, JW91]. The core objective has been to develop scheduling algorithms suited for certain goals, sometimes with provable properties.

8.1.1 Seek-reducing scheduling algorithms

Seek reduction is often crucial for extracting even acceptable throughput for disk intensive applications. The Shortest Positioning-Time First (SPTF) policy greedily aims to improve disk throughput by always scheduling the available request that has the minimum head repositioning overhead time [WGP94]. This policy has two drawbacks: firstly, this policy can incur huge maximum response times due to starvation of old requests. Several variants have been proposed to address this problem. For example, Grouped Shortest Positioning-Time First (GSPTF) divides the disk into cylinder groups, and applies SPTF in each group before moving on to the next [SCO90]. Aged Shortest Positioning-Time First (ASPTF) applies an aging function to the times
computed in SPTF. When a request remains pending for too long, it automatically becomes more likely to get serviced. There are many aging strategies, corresponding to characteristics of the response time bound. ASPTF generally demonstrates remarkable performance, both in terms of high throughput and bounded maximum response time [WGP94, JW91, SCO90].

The second problem with the above algorithms is their requirement for intricate knowledge of the disk layout, in order to predict future head positions and access times accurately. An appreciable component of access time is accountable to rotational latencies, and these are tricky and CPU-intensive to estimate [JW91, HC00]. A popular disk-independent simplification is to schedule requests based on logical block number (LBN) differences, used as an approximation to head repositioning time. Though there can be significant performance loss (up to 18\% in practice) in this approach, the remarkable ease and generality of implementation often warrants its deployment.

An example of these LBN-based scheduling policies is Shortest Seek First (SSF), which provides high throughput, but with possible starvation. A popular and widely deployed variant in UNIX-based systems is Circular Look (C-LOOK), also known as the Elevator algorithm. The head traverses the disk in only one direction, servicing all the available requests on its way. On reaching the last such request, the head moves quickly back to the first available request, and starts over. This algorithm is equally fair towards requests directed at various locations on the disk. It avoids starvation, but can incur huge response times (even minutes) if a process several issues many requests and forces a large portion of the disk to be read [SCO90]. Yet this policy provides fairly good throughput and usually low response time, and is implemented in FreeBSD and Linux.
8.1.2 Proportional-share scheduling algorithms

Quality of service schedulers are increasingly gaining prominence in modern systems. Accurate proportional-share schedulers are required for various high-level quality of service systems, like using reservation domains to isolate co-hosted websites [BGÖS98], and performing admission control to guarantee predictable performance of web servers [AID01]. We examine a brief taxonomy of proportional-share scheduling algorithms.

A proportional-share scheduler is expected to allocate resources between resource principals, in proportion to some assigned shares. Generalized processor sharing (GPS) [PG93] is an idealized "fluid" model that theoretically achieves this goal at all times. However, requests are not infinitesimally divisible, and service can be provided only to one principal at a time. These two intrinsic inexactnesses drive the need for approximations like packet-by-packet GPS (PGPS) [PG93], Weighted fair queueing (WFQ) [DKS89] and VirtualClock [Zha91] in the network packet scheduling domain. The first two are based on a notion of global virtual time, whereas VirtualClock considers per-stream time (and has the problem of an inactive stream later monopolizing link usage [PG93]).

Similar proportional-share CPU scheduling policies have been proposed, with subtle differences. Start-time fair queueing (SFQ) [GGV96] is an application of WFQ to a hierarchical multimedia scheduler. The Stride scheduler [Wal95, WW95] uses methods similar to VirtualClock, and uses the flexible ticket abstraction to manage allocations. Move-to-rear list scheduling (MTR-LS) [BGÖS97] simultaneously guarantees cumulative service, delay bound and fairness. (Cumulative service is shown to be more meaningful on a realistic system with multiple resources to manage. In this context, [BBG+99] mentions how managing each resource in isolation is not realistic due to "closed-loop" behaviour of most applications; this slightly relates to deceptive idleness caused by resource contention, as described in section 3.2). Yet-another fair queueing (YFQ) [BBG+99] is a disk scheduling variant to WFQ, and performs novel
disk-specific optimizations like batching and overlapping.

Lottery scheduling [WW94] is a randomized algorithm that allocates resources to principals on a coarse timescale. A unit of resource is assigned to a principal with probability proportional to its assigned share.

8.2 Improved I/O processing outside the scheduling policy

Though scheduling policies have been analyzed in great detail, relatively less attention has gone into examining the interface between the scheduler and the rest of the operating system. For example, Seltzer et al. [SCO90] perform a detail simulation study of various seek reducing schedulers, under the premise that a busy system generally has long and bursty disk queues. Our work realizes that these requests may not be of the right kind, due to the synchronous manner in which applications generate them. Deceptive idleness thus originates at this interface, and is closely related to other techniques implemented in the I/O subsystem.

For write requests, IRIX implements I/O pacing to prevent programs from saturating the system’s I/O facilities. It enforces per-file high and low water marks on the number of queued requests. The low water mark here is to buffer write requests and increase opportunities for seek optimization; this can be viewed as a logical counterpart of NWCS for write requests.

Also in the context of efficiently handling asynchronous requests, freeblock scheduling [LSG+00] has been proposed to increase media bandwidth utilization. It dictates head movement paths so as to service asynchronous requests approximately enroute to the synchronous ones. This system bears a logical similarity to NWCS; both improve the scheduling policy by making use of information about request characteristics, acquired at the scheduler-kernel interface.

Like deceptive idleness, a different kind of scheduler starvation has been noted by [JW91] in the context of the Aged-SPTF scheduler. The condition arises if aging requests are prioritized abruptly at some age threshold, and if the rate of incoming
requests exceeds service rate. Every scheduler choice becomes forced due to prioritization rather than SPTF, and the scheduler degenerates to FCFS. This is a more obvious phenomenon than deceptive idleness, and the solution clearly involves smooth aging.

Our NWCS solution adaptively determines application behaviour with respect to synchronous request issue. Understanding application behaviour in various aspects of the disk subsystem can yield significant benefits, and such methods are gaining prominence. For example, adaptive block rearrangement [AS93] co-locates frequently referenced blocks, thus improving seek performance.

Filesystem prefetching [SSS99] is a well-researched area, and for moderately regular workloads, asynchronous prefetch can hope to transparently eliminate deceptive idleness and bring about improved performance. To improve the feasibility of prefetch, techniques such as transparent hints by speculative execution [PGG+95] have been proposed. In comparison, our NWCS approach generally achieves slightly lower performance than prefetched I/O, but is feasible in more situations due to weaker demands on predictive power.

8.3 Other scheduling domains

We take a brief look at CPU and network interface scheduling, to identify some interesting phenomena related to deceptive idleness.

8.3.1 CPU scheduling

CPU scheduling is preemptible, so there is no analog of deceptive idleness in this domain. There is, however, the equivalent of seek optimization by scheduling requests from the same process. Namely, Ousterhout has proposed gang scheduling to reduce context switching overhead, by scheduling groups of threads together [Ous82]. Various studies of cache affinity on processor scheduling have also been conducted [VZ91].

On a different note, non-work-conserving CPU schedulers of various types have
been clearly established as a favourable mechanism for handling bursty workloads. Differentiated, prioritized levels of service in a webserver can be provided by scheduling incoming requests from within the kernel. Even when a CPU is idle, not servicing a low-priority request can prevent it from choking higher-priority requests that arrive soon after. We can enforce stricter priorities under bursty load. [ADMC98].

A CPU scheduler can be non-work-conserving in a different way, by keeping a few CPUs idle for handling unexpected and bursty workloads. A detailed simulation-driven analysis [RSSD95] has shown its effectiveness for non-scalable workloads, possibly with high variance in arrival rate and execution time.

8.3.2 Network packet scheduling

Rate-based flow control by network packet scheduling has traditionally paced the way for concepts like class-based queueing. This domain is non-preemptible, but there is no real reason to expect deceptive idleness. The high bandwidth-delay product forces applications to always maintain windows of outstanding packets, due to which the scheduler never starves. There is no equivalent of context-switching overhead, so a scheduler has no reason to batch packets.

Interestingly, there is an opposite effect here. Consider WFQ [DKS89] scheduling between say 11 active queues, one of which has a share of 10 and the others have a share of 1 each. Then WFQ will typically schedule ten requests from the first queue, followed by one request from each of the other queues. This leads to periodic bursts of packets, which is undesirable for network congestion control etc. To remedy this, Worst-case Fair Weighed Fair Queueing (WF²Q) is a work-conserving scheduler that interleaves packets even more than WFQ does [BZ96]. This can be regarded as an optimization in exactly the opposite direction to NWCS, which aggregates disk requests for seek optimization.

Non-work-conserving packet schedulers were originally unpopular because of sub-optimal average performance. However, integrated services networks require end-to-
end delay bounds, which is difficult to provide under bursty workloads. Zhang and Knightly use non-work-conserving schedulers that hold packets in the network, and simulate the original traffic stream. In effect, this approach substantially protects delay bounds from bursty traffic [Zha95, ZK96].
Chapter 9

Conclusion

This thesis has isolated the problem of deceptive idleness, and analyzed its impact on seek reducing and proportional-share schedulers. It has clearly demonstrated the benefits of using a non-work-conserving disk scheduler to solve deceptive idleness, and thus obtain significant improvements in throughput and adherence to quality of service objectives. This includes separate heuristics for seek reducing and proportional-share schedulers, to address their individual needs. This solution smoothly augments various workarounds like application and kernel level prefetching, and practically never causes degradation below a traditional system. It is easy to implement, and suited for incorporation into general-purpose operating systems.

Finally, this thesis has evaluated this solution for a variety of workloads. Microbenchmarks characterize the intrinsic properties of this solution, and real workloads test its effectiveness in more realistic circumstances. The Apache webserver delivers 56% and 16% more throughput for two configurations. The Andrew Benchmark runs faster by 8% (54% for the read-intensive phase). Variants of the TPC-B database benchmark exhibit improvements between 4% and 60%. Proportional-share schedulers become empowered to efficiently deliver application-desired proportions. All this, with almost no overhead.
Appendix: Detailed NWCS pseudocode

This appendix summarizes the implementation of the NWCS waiting mechanism, and separate heuristics for the SPTF (and C-LOOK) and Stride schedulers. This should facilitate easy implementation and deployment of NWCS in current operating systems. This does not include pseudocode for measuring disk characteristics and building an approximate positioning time calculator. For even greater detail, please refer to our prototype implementation. The source code is made publicly available on the project webpage, hosted at [ID01a].

I. Generic NWCS scheduling framework

```c
// sched_enqueue,dequeue,finish,collect_stats,evaluate
// are the scheduler-specific functions described later

enum STATE { WAIT = -1, IDLE = 0, BUSY = 1,2,3,... } state = IDLE;
integer pending = 0;
boolean expired = false;

// these four functions should set the higher interrupt priority level
// of the timer (spliclock or spilhigh), rather than of disk I/O (spibio)

issue(new):
// gets invoked by upper layers of the kernel
// state == IDLE or BUSY or WAIT

if (new.is_read_request) sched_collect_stats(new);
sched_enqueue(new);
pending++;

if (new.is_read_request && (new.is_sync_request || state == WAIT))
    new.proc.blocked = true; // little hope in waiting for this process

if (state == IDLE || state == WAIT) schedule();
```
schedule():
    // internal function. sometimes takes a request from the pool and
    // services it; otherwise just returns with the timeout activated.
    // called with pending > 0

    static request last, next;   // retain values across calls

    if (expired)
    |    // state = WAIT
    |    expired = false;
    |    next = sched_choose(); // need this for SCSI only
    else if (last.proc.blocked)
    |    // state = WAIT
    |    next = sched_choose();
    else
    |    // state = IDLE or WAIT, or BUSY if SCSI
    |    next = sched_choose()
    |    duration = next.is_read_request ?
    |        sched_evaluate(last, next) : 0;
    |    if (duration > 0)
    |        if (state == IDLE)
    |            state = WAIT;
    |        else
    |            // do nothing; in particular, don't retrIGGER timeout
    |        return; // "next" contains the chosen candidate
    next = sched_choose();   // probably need this for SCSI
    sched_dequeue(next);
    pending--;
    if (state == WAIT) { cancel_timeout(); state = IDLE; }
    state++; // idle->busy, busy->busier
    last = next; next = NULL;
    PERFORM_DISK_IO(last);  // run request on disk

timeout_expire():
    // gets called by timeout code, in interrupt context
    // state == WAIT, pending > 0, next != NULL
    expired = true;
    schedule();

finish(req):
    // gets called by disk driver when current finishes execution
    // state == BUSY
    state--; // busy:1->idle, busier->busy
    sched_finish(req);
    req.proc.prev_finish_time = NOW;
    if (pending) schedule();
II. NWCS heuristic for SPTF

sched_collect_stats(new):

    // every process has:
    //   buckets:[0..N] with N = 30 (i.e. least count = 500ms)
    //   expected_thinktime, expected_most_thinktime
    //   expected_seek_distance, expected_seek_direction
    //   expected_positioning_time

    if (req \in scheduler_queue
       && req.is_read_request && req.proc == new.proc)
        thinktime = 0
    else
        thinktime = min(15ms, NOW - new.proc.prev_finish_time);
        // the process issuing "new" computes for some time
        // between its finish time for previous request and NOW

    loop:i:(0..N) new.proc.bucket[i] *= 0.9    // decay all buckets
    new.proc.bucket[(thinktime/15ms) * N] += 1; // increment this bucket
    new.proc.expected_thinktime = median(new.proc.buckets)
    new.proc.expected_most_thinktime = 95percentile(new.proc.buckets)

    seekdist = abs(new.location - new.proc.prev_location)
    seekdirection = (new.location > new.proc.prev_location) ? 1 : 0;
    new.proc.prev_location = new.location; // update prev_location

    new.proc.expected_seek_distance *= 0.74
    new.proc.expected_seek_distance += seekdist

    new.proc.expected_seek_direction *= 0.74    // used by C-LOOK heuristic
    new.proc.expected_seek_direction += seekdirection

    new.proc.expected_positioning_time =
        calculate_postime(new.proc.expected_seek_distance)
        // we use an approximation of positioning time based on seek
        // distance; an exact version would need some more arguments

sched_evaluate(last, next):

    next_seek_distance = abs(next.location - last.location)
    next_seek_direction = next.location > last.location;
    next_positioning_time = calculate_postime(next_seek_distance)

    // ADDITIONAL CHECK FOR THE C-LOOK HEURISTIC ONLY
    if (next_seek_direction == 1 /* ahead */
    || && last.proc.expected_seek_direction < 0.3 /* behind */)
III. NWCS heuristic for Stride

sched_collect_stats(new): reduced version of (or same as) that for SPTF

sched_evaluate(last, next): // PROCESS VERSION

```
minclock = \min\{\ proc[i].clock, \text{ where } (proc[i].pending > 0) \} 
elapsed = NOW - last.proc.prev_finish_time;
if (last.proc.pending == 0)
  \&\& last.proc.clock < minclock 
  \&\& max(last.proc.expected_thinktime - elapsed, 0) < 3ms)
  return max(last.proc.expected_thinktime - elapsed, 0)
else
  return 0;
```

sched_evaluate(last, next): // RESOURCE CONTAINER VERSION

```
minclock = \min\{\ rc[i].clock, \text{ where } (rc[i].pending > 0) \} 
elapsed = NOW - last.proc.prev_finish_time;
if (last.proc.rc.pending == 0)
  \&\& last.proc.rc.clock < minclock 
  \&\& max(last.proc.expected_thinktime - elapsed, 0) < 3ms)
  return max(last.proc.expected_thinktime - elapsed, 0)
else
  return 0;
```
Bibliography


